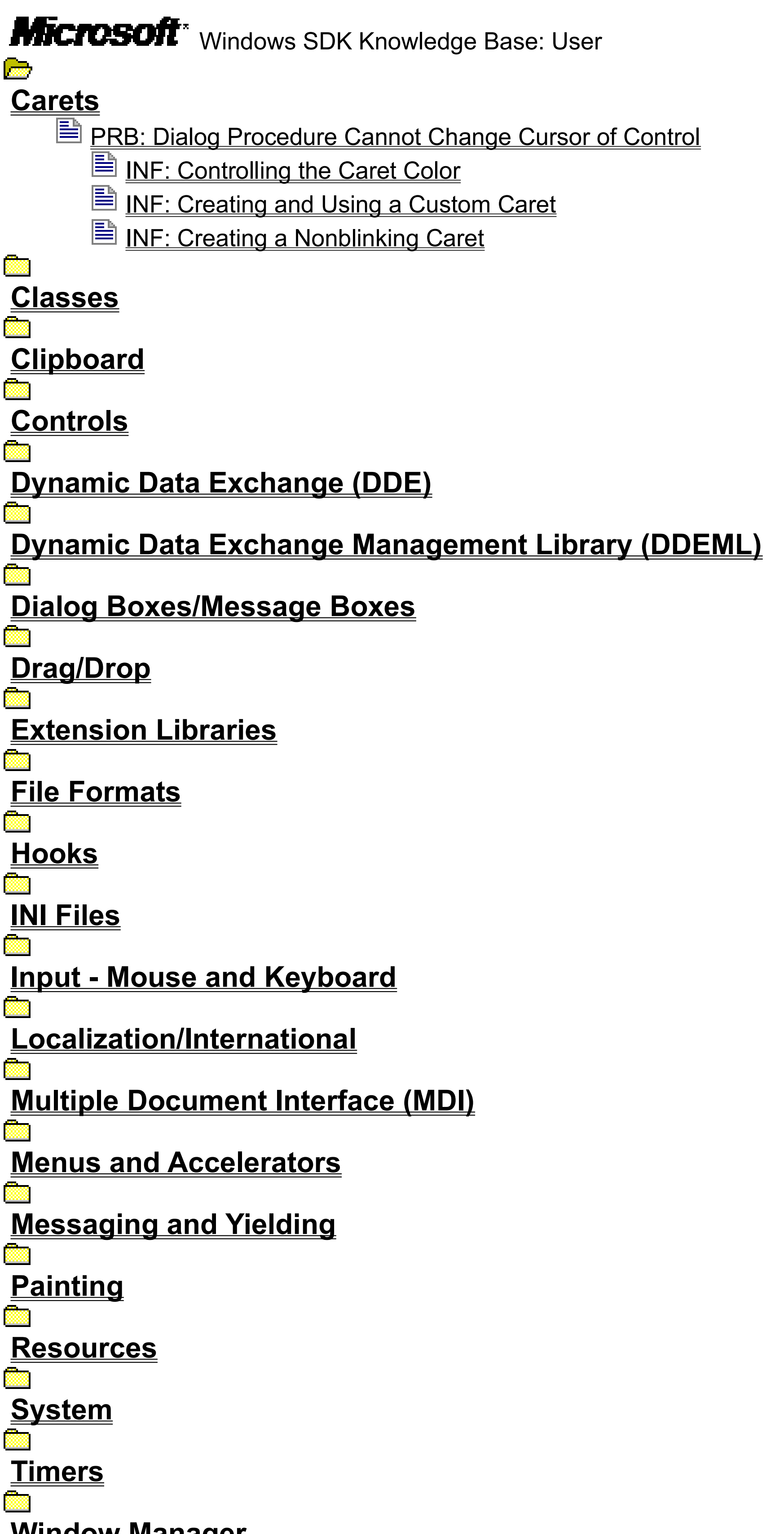
Microsoft Windows SDK Knowledge Base: User
Prepared 11/17/93

Carets
Classes
Clipboard
Controls
Dynamic Data Exchange (DDE)
Dynamic Data Exchange Management Library (DDEML)
Dialog Boxes/Message Boxes
Drag/Drop
Extension Libraries
File Formats
Hooks
INI Files
Input - Mouse and Keyboard
Localization/International
Multiple Document Interface (MDI)
Menus and Accelerators
Messaging and Yielding
Painting
Resources
System
Timers
Window Manager

THE INFORMATION IN THE MICROSOFT KNOWLEDGE BASE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. MICROSOFT DISCLAIMS ALL WARRANTIES EITHER EXPRESSED OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MICROSOFT CORPORATION OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER INCLUDING DIRECT, INDIRECT, INCIDENTAL,

CONSEQUENTIAL, LOSS OF BUSINESS PROFITS, OR SPECIAL DAMAGES, EVEN IF MICROSOFT CORPORATION OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES SO THE FORGOING EXCLUSION OR LIMITATION MAY NOT APPLY.


Windows SDK Knowledge Base: User

Carets

- PRB: Dialog Procedure Cannot Change Cursor of Control
  - INF: Controlling the Caret Color
  - INF: Creating and Using a Custom Caret
  - INF: Creating a Nonblinking Caret

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

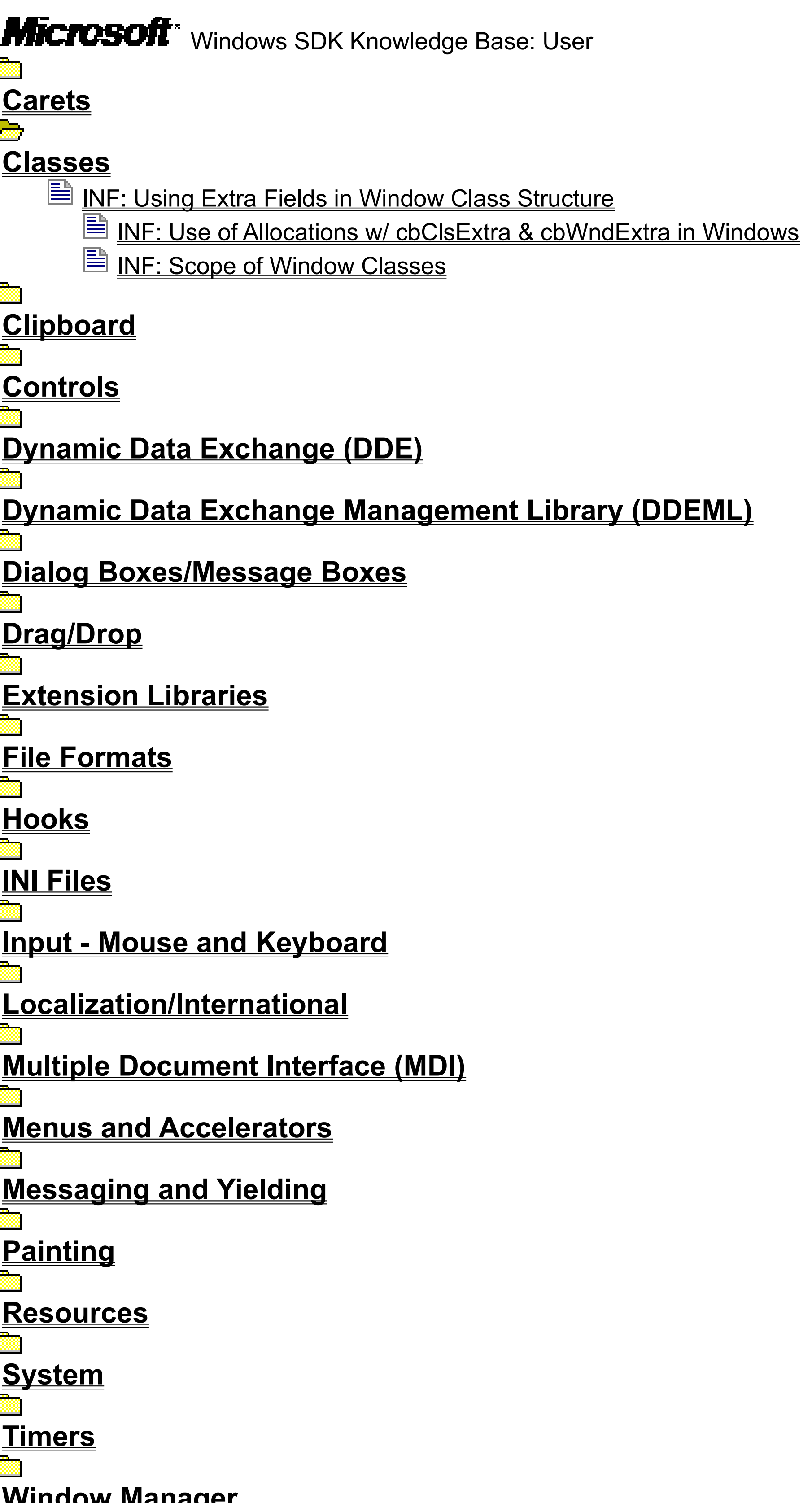
Painting

Resources

System

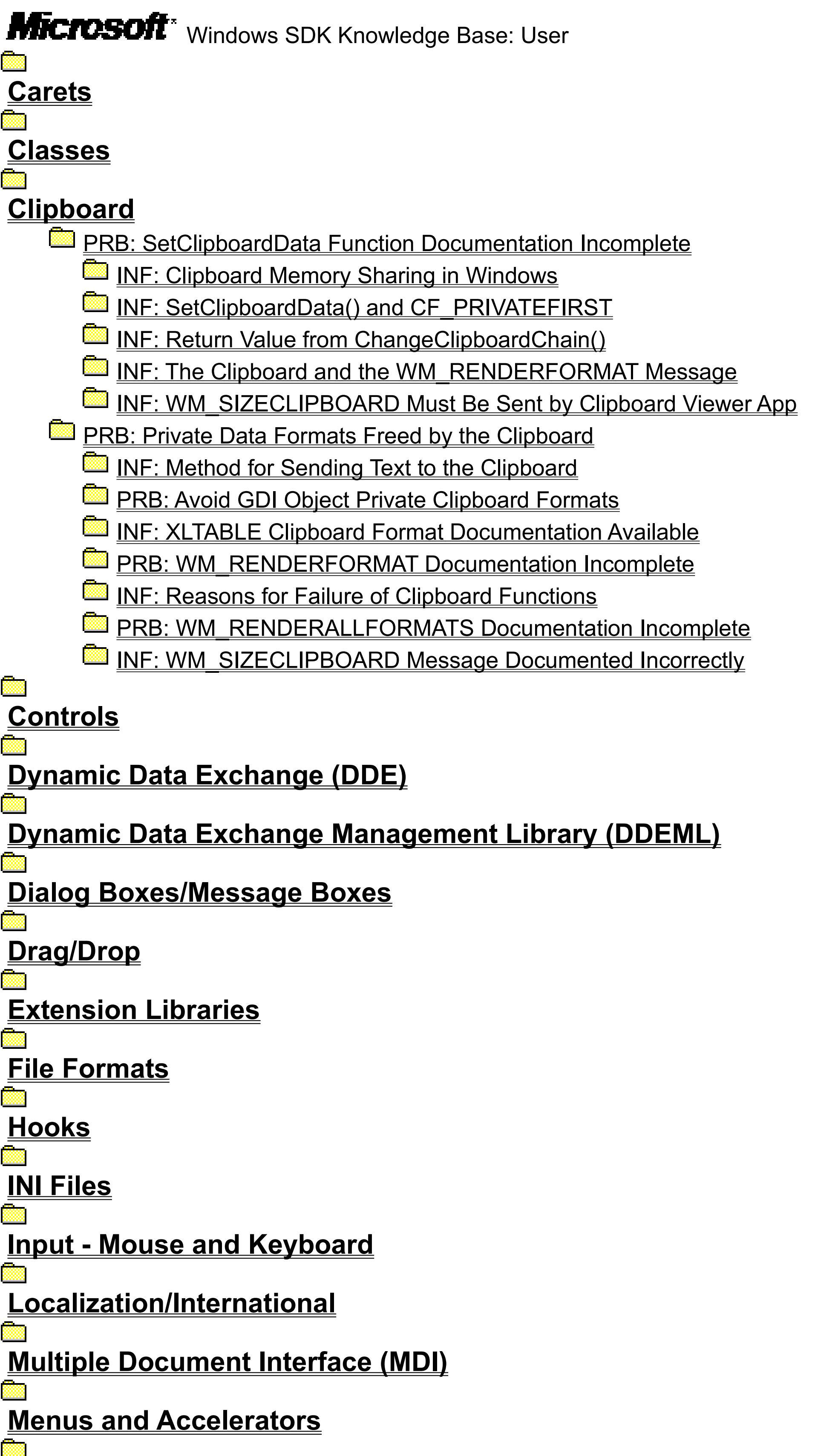
Timers

Window Manager



Microsoft Windows SDK Knowledge Base: User
Carets
Classes
INF: Using Extra Fields in Window Class Structure
INF: Use of Allocations w/ cbClsExtra & cbWndExtra in Windows
INF: Scope of Window Classes
Clipboard
Controls
Dynamic Data Exchange (DDE)
Dynamic Data Exchange Management Library (DDEML)
Dialog Boxes/Message Boxes
Drag/Drop
Extension Libraries
File Formats
Hooks
INI Files
Input - Mouse and Keyboard
Localization/International
Multiple Document Interface (MDI)
Menus and Accelerators
Messaging and Yielding
Painting
Resources
System
Timers
Window Manager

Microsoft Windows SDK Knowledge Base: User



- **Carets**
- **Classes**
- **Clipboard**
  - PRB: SetClipboardData Function Documentation Incomplete
    - INF: Clipboard Memory Sharing in Windows
    - INF: SetClipboardData() and CF_PRIVATEFIRST
    - INF: Return Value from ChangeClipboardChain()
    - INF: The Clipboard and the WM_RENDERFORMAT Message
    - INF: WM_SIZECLIPBOARD Must Be Sent by Clipboard Viewer App
  - PRB: Private Data Formats Freed by the Clipboard
    - INF: Method for Sending Text to the Clipboard
    - PRB: Avoid GDI Object Private Clipboard Formats
    - INF: XLTABLE Clipboard Format Documentation Available
    - PRB: WM_RENDERFORMAT Documentation Incomplete
    - INF: Reasons for Failure of Clipboard Functions
    - PRB: WM_RENDERALLFORMATS Documentation Incomplete
    - INF: WM_SIZECLIPBOARD Message Documented Incorrectly
- **Controls**
- **Dynamic Data Exchange (DDE)**
- **Dynamic Data Exchange Management Library (DDEML)**
- **Dialog Boxes/Message Boxes**
- **Drag/Drop**
- **Extension Libraries**
- **File Formats**
- **Hooks**
- **INI Files**
- **Input - Mouse and Keyboard**
- **Localization/International**
- **Multiple Document Interface (MDI)**
- **Menus and Accelerators**

Messaging and Yielding
Painting
Resources
System
Timers
Window Manager

Windows SDK Knowledge Base: User

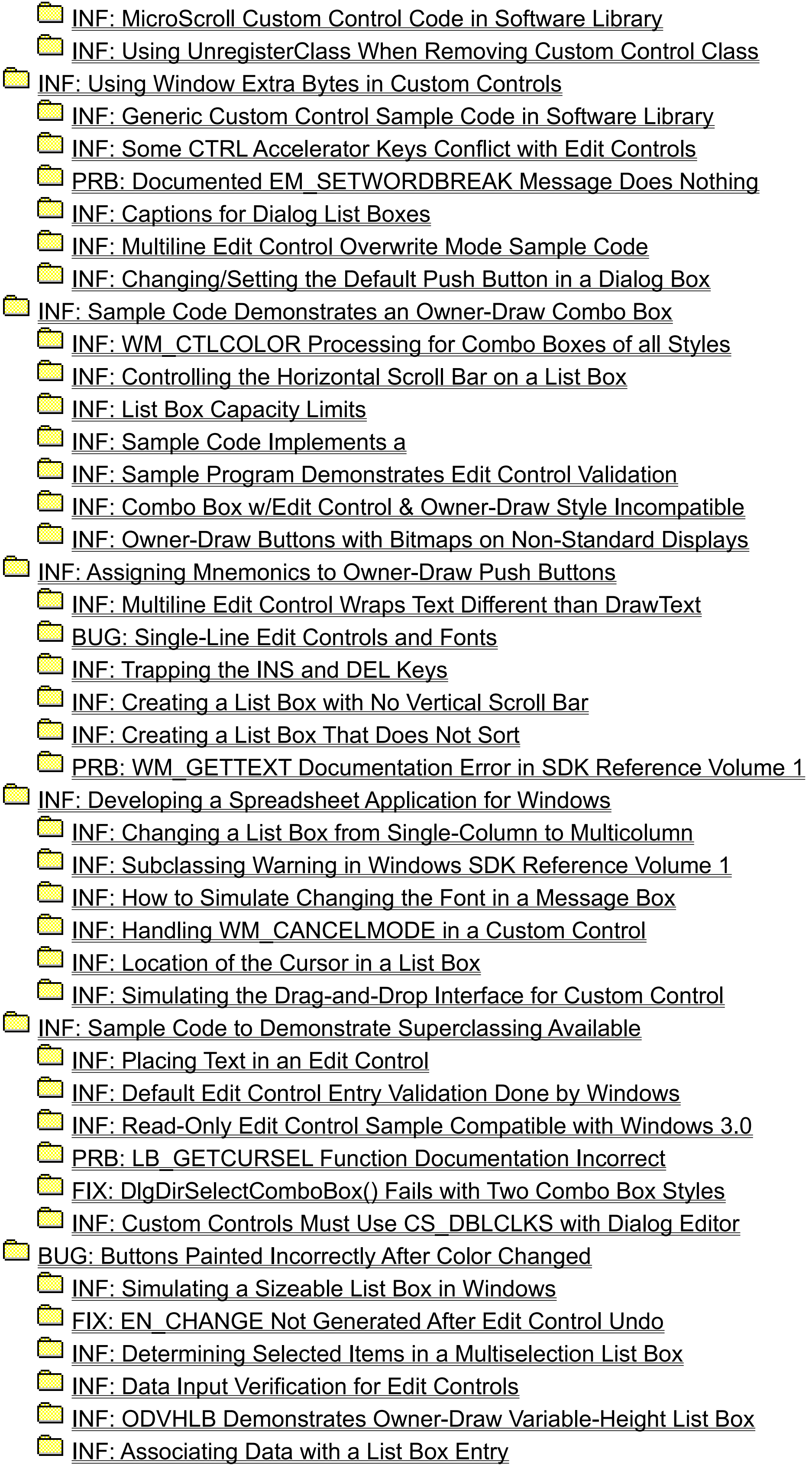
Carets

Classes

Clipboard

Controls

- BUG: WM_CTLCOLOR Message Not Sent to Combo Box
  - INF: Placing a Caret After Edit-Control Text
  - INF: Placing Captions on Control Windows
  - INF: Right Justifying Numbers in a Windows 3.0 List Box
- INF: Creating a List Box Without a Scroll Bar
  - INF: Retrieve Line Number from Edit Control Sample Code
  - INF: Owner-Draw: 3-D Push Button Made from Bitmaps with Text
  - FIX: Problem Adding Horizontal Scroll Bar to List Box
  - INF: Multicolumn List Boxes in Microsoft Windows
  - INF: LBS_STANDARD List Box Style Documented Incorrectly
  - INF: Showing the Beginning of an Edit Control after EM_SETSEL
- INF: Changing Text Alignment in an Edit Control Dynamiclly
  - FIX: Dialog Editor Will Not Change Custom Control Style
  - INF: Limiting the Number of Entries in a List Box
  - INF: Freeing Resources Used by a Multiline Edit Control
  - FIX: Edit Notification Codes Never Received
  - INF: Determining the Number of Visible Items in a List Box
  - INF: Size Limits for a Multiline Edit Control
- FIX: Cannot Change Single-Line Edit Control Color
  - INF: Overcoming the 64 Kilobyte Limit for List Box Data
  - INF: The Parts of a Windows Combo Box and How They Relate
  - INF: Combo Box Case Where GetDlgItemText() Parameter Ignored
  - INF: Action of Static Text Controls with Mnemonics
  - PRB: Multiline Edit Controls UAE If Resizing Changes LineCount
- PRB: Messages Processed by Single & Multiline Edit Controls
  - PRB: CBN_SELCHANGE Documented Incorrectly
  - INF: Processing CBN_SELCHANGE Notification Message
  - INF: Sample Code Demonstrates Changing Size of an Edit Control
  - INF: EM_SETTABSTOPS Does Not Free All of Local Heap Memory
  - INF: Preventing Screen Flash During List Box Multiple Update
  - INF: Changing the Color of an Edit Control
- INF: Setting Tab Stops in a Windows 3.0 List Box
  - INF: Multiline Edit Control Does Not Show First Line
  - INF: Implementing a Read-Only Edit Control In Windows
  - PRB: Custom Control Documentation Errors
  - FIX: No Focus Rect on List Box at Dialog Creation
  - INF: Controlling the Horizontal Scroll Bar on a List Box



  - INF: MicroScroll Custom Control Code in Software Library
  - INF: Using UnregisterClass When Removing Custom Control Class
- INF: Using Window Extra Bytes in Custom Controls
  - INF: Generic Custom Control Sample Code in Software Library
  - INF: Some CTRL Accelerator Keys Conflict with Edit Controls
  - PRB: Documented EM_SETWORDBREAK Message Does Nothing
  - INF: Captions for Dialog List Boxes
  - INF: Multiline Edit Control Overwrite Mode Sample Code
  - INF: Changing/Setting the Default Push Button in a Dialog Box
- INF: Sample Code Demonstrates an Owner-Draw Combo Box
  - INF: WM_CTLCOLOR Processing for Combo Boxes of all Styles
  - INF: Controlling the Horizontal Scroll Bar on a List Box
  - INF: List Box Capacity Limits
  - INF: Sample Code Implements a
  - INF: Sample Program Demonstrates Edit Control Validation
  - INF: Combo Box w/Edit Control & Owner-Draw Style Incompatible
  - INF: Owner-Draw Buttons with Bitmaps on Non-Standard Displays
- INF: Assigning Mnemonics to Owner-Draw Push Buttons
  - INF: Multiline Edit Control Wraps Text Different than DrawText
  - BUG: Single-Line Edit Controls and Fonts
  - INF: Trapping the INS and DEL Keys
  - INF: Creating a List Box with No Vertical Scroll Bar
  - INF: Creating a List Box That Does Not Sort
  - PRB: WM_GETTEXT Documentation Error in SDK Reference Volume 1
- INF: Developing a Spreadsheet Application for Windows
  - INF: Changing a List Box from Single-Column to Multicolumn
  - INF: Subclassing Warning in Windows SDK Reference Volume 1
  - INF: How to Simulate Changing the Font in a Message Box
  - INF: Handling WM_CANCELMODE in a Custom Control
  - INF: Location of the Cursor in a List Box
  - INF: Simulating the Drag-and-Drop Interface for Custom Control
- INF: Sample Code to Demonstrate Superclassing Available
  - INF: Placing Text in an Edit Control
  - INF: Default Edit Control Entry Validation Done by Windows
  - INF: Read-Only Edit Control Sample Compatible with Windows 3.0
  - PRB: LB_GETCURSEL Function Documentation Incorrect
  - FIX: DlgDirSelectComboBox() Fails with Two Combo Box Styles
  - INF: Custom Controls Must Use CS_DBLCLKS with Dialog Editor
- BUG: Buttons Painted Incorrectly After Color Changed
  - INF: Simulating a Sizeable List Box in Windows
  - FIX: EN_CHANGE Not Generated After Edit Control Undo
  - INF: Determining Selected Items in a Multiselection List Box
  - INF: Data Input Verification for Edit Controls
  - INF: ODVHLB Demonstrates Owner-Draw Variable-Height List Box
  - INF: Associating Data with a List Box Entry

- BUG: Combo Boxes in DS_SYSMODAL Dialog Boxes
  - FIX: Drawing Problem When Group Box Text Changes
  - INF: Graying the Text of a Button or Static Text Control
  - FIX: EM_SETWORDBREAK Not Implemented in Windows 3.0
  - INF: Limit to the Number of Characters Stored in a List Box
  - INF: Determining the Visible Area of a Multiline Edit Control
  - INF: Making a List Box Item Unavailable for Selection
- INF: Buttons and Cursors Documentation and Sample
  - INF: Sample Code to Demonstrate a Button Bar
  - INF: Altering Edit Control Strings in Place May Cause UAE
  - INF: Changing an Edit Control to a Bedit Control at Run Time
  - PRB: Accented Characters in Filename Controls Lose Accents
  - PRB: WM_SETTEXT Ignores EM_LIMITTEXT Edit Control Text Limit
  - INF: Creating the Default Border Around a Push Button
- PRB: Moving or Resizing the Parent of an Open Combo Box
  - INF: Switching Between Single and Multiple List Boxes
  - PRB: Button Styles May Not Be Combined
  - INF: Extending Standard Windows Controls Through Superclassing
  - FIX: Resizing Multiline Edit Control Causes UAE
  - INF: Raising Text Size Limit for Edit Controls
  - INF: Subclassing Sample Code Available in Software Library
- INF: DlgDirList on Novell Drive Doesn't Show Double Dots [..]
  - INF: Safe Subclassing
  - INF: EM_SETSEL wParam Not Used in Single Line Edit Controls
  - SAMPLE: Moving an Item in a List Box Using Drag and Drop
  - SAMPLE: Subclassing VBX Controls with MFC 2.0
  - INF: SetParent and Control Notifications
  - INF:Unselecting Edit Control Text at Dialog Box Initialization

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

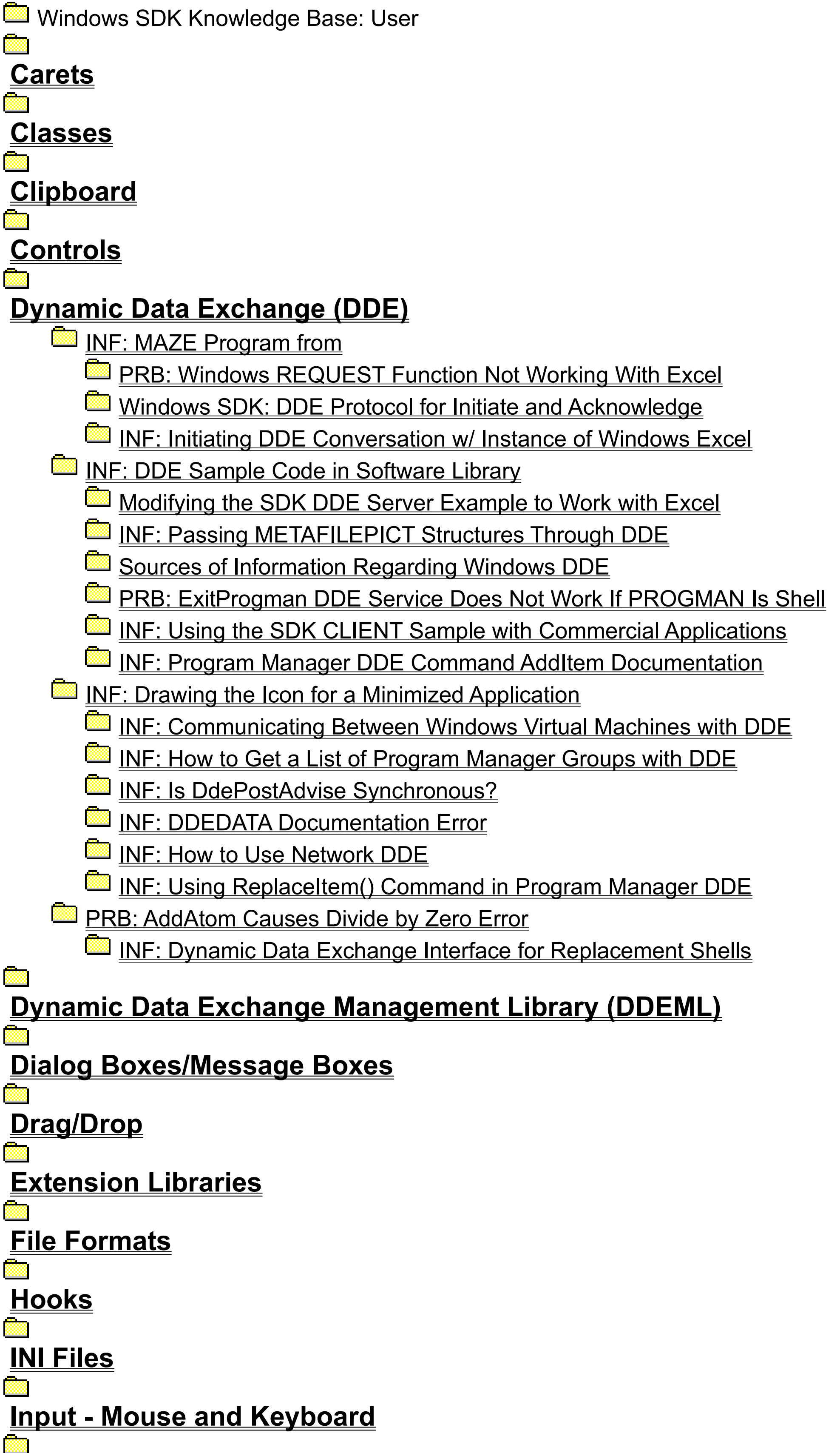
File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International
Multiple Document Interface (MDI)
Menus and Accelerators
Messaging and Yielding
Painting
Resources
System
Timers
Window Manager



- Windows SDK Knowledge Base: User
- **Carets**
- **Classes**
- **Clipboard**
- **Controls**
- **Dynamic Data Exchange (DDE)**
  - INF: MAZE Program from
    - PRB: Windows REQUEST Function Not Working With Excel
    - Windows SDK: DDE Protocol for Initiate and Acknowledge
    - INF: Initiating DDE Conversation w/ Instance of Windows Excel
  - INF: DDE Sample Code in Software Library
    - Modifying the SDK DDE Server Example to Work with Excel
    - INF: Passing METAFILEPICT Structures Through DDE
    - Sources of Information Regarding Windows DDE
    - PRB: ExitProgman DDE Service Does Not Work If PROGMAN Is Shell
    - INF: Using the SDK CLIENT Sample with Commercial Applications
    - INF: Program Manager DDE Command AddItem Documentation
  - INF: Drawing the Icon for a Minimized Application
    - INF: Communicating Between Windows Virtual Machines with DDE
    - INF: How to Get a List of Program Manager Groups with DDE
    - INF: Is DdePostAdvise Synchronous?
    - INF: DDEDATA Documentation Error
    - INF: How to Use Network DDE
    - INF: Using ReplaceItem() Command in Program Manager DDE
  - PRB: AddAtom Causes Divide by Zero Error
    - INF: Dynamic Data Exchange Interface for Replacement Shells
- **Dynamic Data Exchange Management Library (DDEML)**
- **Dialog Boxes/Message Boxes**
- **Drag/Drop**
- **Extension Libraries**
- **File Formats**
- **Hooks**
- **INI Files**
- **Input - Mouse and Keyboard**

Localization/International
Multiple Document Interface (MDI)
Menus and Accelerators
Messaging and Yielding
Painting
Resources
System
Timers
Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

- PRB: Using HSZ in AFXEXT.H and DDEML.H
  - INF: WM_DDE_EXECUTE Message Must Be Posted to a Window
  - INF: Sample Code Demonstrates DDEML with Metafiles
- INF: Description of the DDEML Error Codes
  - INF: DDEML CONVINFO Structure, wConvst Field Description
  - INF: Sample Application Demonstrates Using DDEML
  - INF: Freeing Memory in a DDEML Server Application
  - INF: Executing Excel Functions with Return Values Using DDE
  - INF: Freeing Memory for Transactions in a DDEML Client App
  - INF: CONVINFO Data Structure wStatus Field Description
- PRB: GP Fault in DDEML from XTYP_EXECUTE Timeout Value
  - INF: Application Can Allocate Memory with DdeCreateDataHandle
  - PRB: DdeUnaccessData Function Documented Incorrectly
  - INF: Software Library Has DDE Management Library Information
  - INF: Do Not Forward DDEML Messages from a Hook Procedure
  - INF: Sample: DDEML Samples Using Microsoft Foundation Classes
  - SAMPLE: Shell DDE Using DDEML
- PRB: DDESPY GP Faults Upon Return of CBR_BLOCK
  - INF: DdeCreateStringHandle() lpszString param Docerr
  - INF: Calling DdePostAdvise() from XTYP_ADVREQ
  - INF: Changes Between Win 3.1 and WFW 3.1 Versions of DDEML
  - INF: XTYP_EXECUTE and its Return Value Limitations
  - INF: Obtaining Group/Item Info from ProgMan Using DDEML
  - PRB: DDEML Fails to Call TranslateMessage() in its Modal Loop
- INF: Returning CBR_BLOCK from DDEML Transactions

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

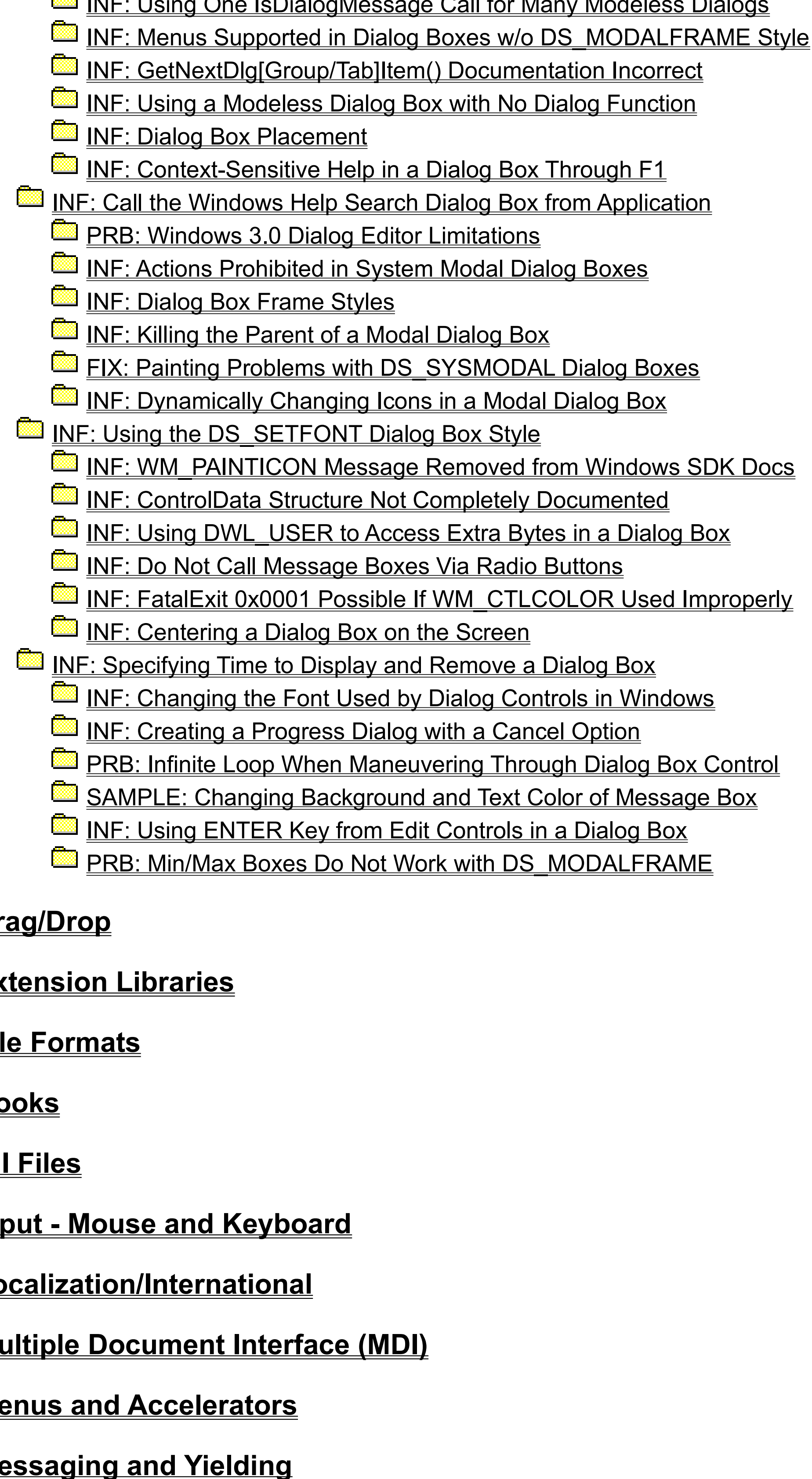
Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

- INF: Using Main Window Edit Menu with Dialog Box Edit Controls
  - INF: Modal Dialog Child of Modeless Dialog Box Sample Code
  - INF: ENTER & TAB Keys in a Dialog Box Multiline Edit Control
- INF: Using a Fixed-Pitch Font in a Dialog Box
  - INF: Efficiency of Using SendMessage Versus SendDlgItemMessage
  - INF: Expanding the Size of a Dialog Box
  - PRB: Dialog Box with Edit Control Cannot Be Created
  - INF: Sample Code Demonstrates Using Dialog Box Templates
  - INF: Scrolling Dialog Box Sample Code in Software Library
  - INF: Sample Code Demonstrates an Application Sign-On Screen
- INF: Overlapping Controls Are Not Supported by Windows
  - PRB: BS_GROUPBOX-Style Child Window Background Painting Wrong
  - INF: How Dialog Box Functions Return Values Indicate Failure
  - PRB: Dialog Box and Parent Window Disabled
  - INF: Possible Causes of Dialog Box Creation Failure
  - INF: Windows Dialog-Box Style DS_ABSALIGN
  - INF: Creating a Multiple Line Message Box
- INF: Obtaining Index of Focused Item in Multi-Select List Box
  - INF: Sample Code Demonstrates Creating Dialog Box in DLL
  - INF: Do Not Use MB_NOFOCUS Flag with MessageBox Function
  - INF: Device Independent Way to Use Dialog Box as Main Window
  - INF: Default/Private Dialog Classes, Procedures, DefDlgProc
  - INF: Using the WM_GETDLGCODE Message
  - INF: Sample Code Demonstrates Using Private Dialog-Box Class
- INF: Types of System Modal Message Boxes
  - INF: Clearing a Message Box
  - Using IsDialogMessage to Simulate a Dialog Box
  - PRB: Disabling All Controls in a Dialog Box Hangs Windows
  - INF: Sample Code Demonstrates How to Add Menus to a Dialog Box
  - PRB: Undesirable Interactions Between Dialog Box Types
  - PRB: Vertical Bars Displayed in Message Box, Control Text
- INF: Sample Code Demonstrates Changing Dialog Box Size



    - INF: Using One IsDialogMessage Call for Many Modeless Dialogs
    - INF: Menus Supported in Dialog Boxes w/o DS_MODALFRAME Style
    - INF: GetNextDlg[Group/Tab]Item() Documentation Incorrect
    - INF: Using a Modeless Dialog Box with No Dialog Function
    - INF: Dialog Box Placement
    - INF: Context-Sensitive Help in a Dialog Box Through F1
  - INF: Call the Windows Help Search Dialog Box from Application
    - PRB: Windows 3.0 Dialog Editor Limitations
    - INF: Actions Prohibited in System Modal Dialog Boxes
    - INF: Dialog Box Frame Styles
    - INF: Killing the Parent of a Modal Dialog Box
    - FIX: Painting Problems with DS_SYSMODAL Dialog Boxes
    - INF: Dynamically Changing Icons in a Modal Dialog Box
  - INF: Using the DS_SETFONT Dialog Box Style
    - INF: WM_PAINTICON Message Removed from Windows SDK Docs
    - INF: ControlData Structure Not Completely Documented
    - INF: Using DWL_USER to Access Extra Bytes in a Dialog Box
    - INF: Do Not Call Message Boxes Via Radio Buttons
    - INF: FatalExit 0x0001 Possible If WM_CTLCOLOR Used Improperly
    - INF: Centering a Dialog Box on the Screen
  - INF: Specifying Time to Display and Remove a Dialog Box
    - INF: Changing the Font Used by Dialog Controls in Windows
    - INF: Creating a Progress Dialog with a Cancel Option
    - PRB: Infinite Loop When Maneuvering Through Dialog Box Control
    - SAMPLE: Changing Background and Text Color of Message Box
    - INF: Using ENTER Key from Edit Controls in a Dialog Box
    - PRB: Min/Max Boxes Do Not Work with DS_MODALFRAME

- **Drag/Drop**
- **Extension Libraries**
- **File Formats**
- **Hooks**
- **INI Files**
- **Input - Mouse and Keyboard**
- **Localization/International**
- **Multiple Document Interface (MDI)**
- **Menus and Accelerators**
- **Messaging and Yielding**

Painting
Resources
System
Timers
Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

- INF: Implementing the Drag-Drop Protocol
  - INF: Windows Version 3.00 Drag and Drop Protocol
- INF: Microsoft Drag-Drop Server Strategy

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

- INF: Using Drag-Drop in an Edit Control or a Combo Box
- SAMPLE: Connect Net Drive--a File Manager Extension
  - SAMPLE: Accelerators for File Manager Extensions
  - INF: Explanation of the NEWCPLINFO Structure

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

- INF: Windows Cardfile File Format
- INF: Windows Program Manager Edits GRP Files
  - INF: Windows Paintbrush File Format
  - INF: Group File Format for Windows Program Manager Version 3.0
  - INF: Windows Accessories Binary File Formats
  - INF: Executable-File Header Format
  - INF: Font-File Format
  - INF: Corrections to Program Manager Group File Format Docs
  - INF: Sample Code to Access New EXE File Headers
- INF: Windows 3.1 Card File Format

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System
Timers
Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

- INF: Windows 3.00 Sample Source Code for a Keyboard Filter
  - INF: Sample Code Demonstrates Windows 3.1 WH_MOUSE Hook
  - INF: Sample Code Demonstrates Using a WH_KEYBOARD Hook
  - INF: Sample Code Uses Keyboard Hook to Access Help
  - INF: Cannot Alter Messages with WH_KEYBOARD Hook
  - INF: Correction to JournalRecordProc Documentation
  - INF: Windows Journal Hooks Sample Source Code
  - INF: Importance of Calling DefHookProc()
- PRB: DLL System Hook Function Not Affecting Apps System-Wide
  - INF: How to Stop a Journal Playback
  - INF: Calling SendMessage() Inside a Hook Filter Function
  - PRB: SetWindowsHookEx() Fails to Install Task-Specific Filter
  - INF: Sample Code for WH_CALLWNDPROC and WH_GETMESSAGE Hooks
  - INF: Message Structure Used by WH_CALLWNDPROC Filter Function
  - PRB: Using ToAscii() in Journal Record Hooks
- INF: Determining Message Removal from WH_GETMESSAGE Hook

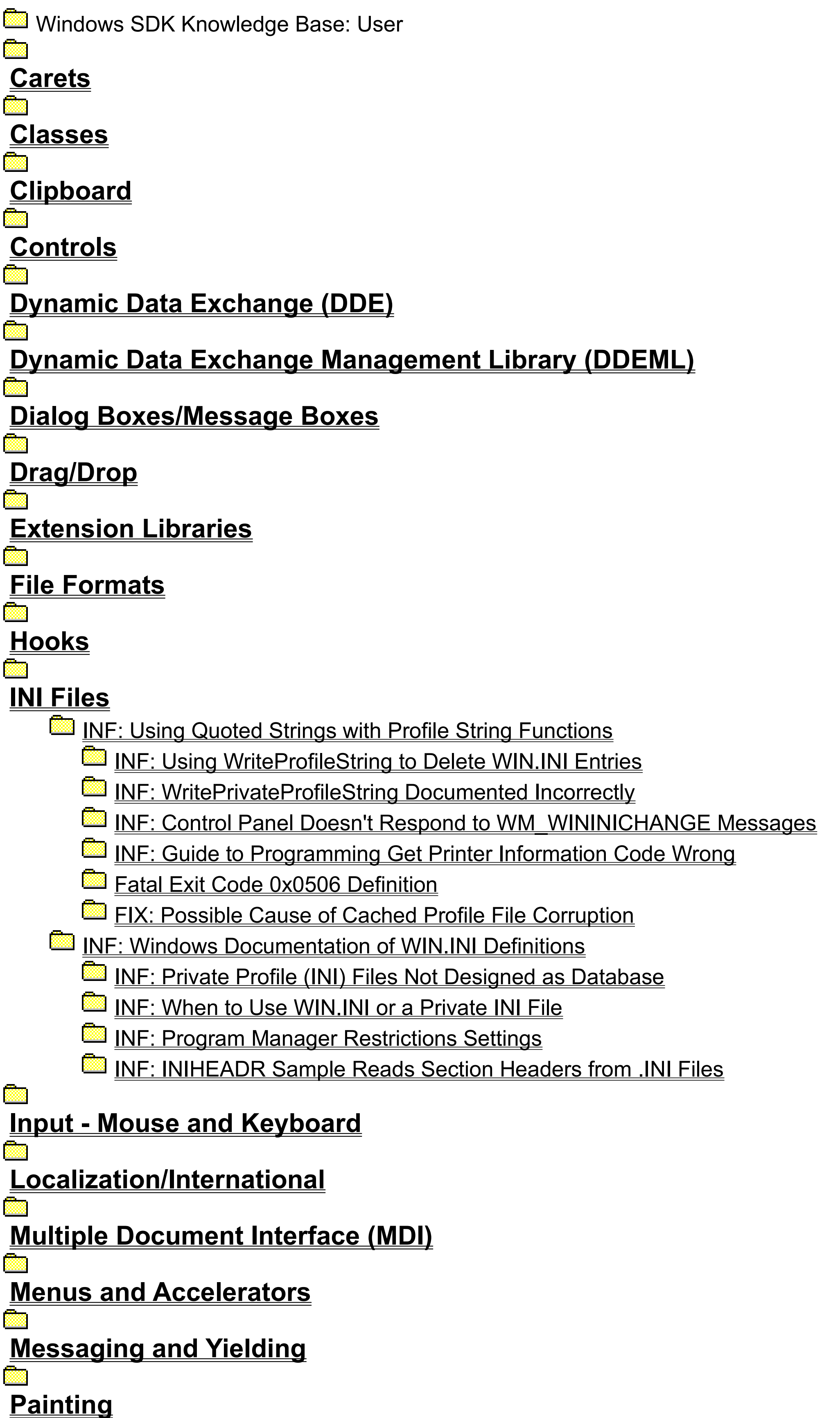
INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators
Messaging and Yielding
Painting
Resources
System
Timers
Window Manager



Windows SDK Knowledge Base: User

- **Carets**
- **Classes**
- **Clipboard**
- **Controls**
- **Dynamic Data Exchange (DDE)**
- **Dynamic Data Exchange Management Library (DDEML)**
- **Dialog Boxes/Message Boxes**
- **Drag/Drop**
- **Extension Libraries**
- **File Formats**
- **Hooks**
- **INI Files**
  - INF: Using Quoted Strings with Profile String Functions
    - INF: Using WriteProfileString to Delete WIN.INI Entries
    - INF: WritePrivateProfileString Documented Incorrectly
    - INF: Control Panel Doesn't Respond to WM_WININICHANGE Messages
    - INF: Guide to Programming Get Printer Information Code Wrong
    - Fatal Exit Code 0x0506 Definition
    - FIX: Possible Cause of Cached Profile File Corruption
  - INF: Windows Documentation of WIN.INI Definitions
    - INF: Private Profile (INI) Files Not Designed as Database
    - INF: When to Use WIN.INI or a Private INI File
    - INF: Program Manager Restrictions Settings
    - INF: INIHEADR Sample Reads Section Headers from .INI Files
- **Input - Mouse and Keyboard**
- **Localization/International**
- **Multiple Document Interface (MDI)**
- **Menus and Accelerators**
- **Messaging and Yielding**
- **Painting**

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

- INF: Return Value of SwapMouseButton() Documented Incorrectly
  - INF: Changing the Mouse Cursor for a Window
  - PRB: Input Focus Lost When Control Returns From Windows Help
  - INF: Detecting Keystrokes While a Menu Is Pulled Down
  - PRWIN9105003: FatalExit() Interacts Through Terminal Only
  - PRB: WM_CHARTOITEM Messages Not Recieved by Parent of List Box
- PRWIN9105004: MapVirtualKey() Maps Keypad Keys Incorrectly
  - INF: Differentiating Between the Two ENTER Keys

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers
Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

- INF: Installed Language, Character Set, and Code Page
  - INF: Writing International Applications for Windows 3.00
  - INF: Extended Characters Different Under Windows
  - PRB: IsCharAlpha Return Value Different Between Versions
  - PRB: Multikey Help Code Incorrect in Windows Tools Manual
- INF: UNINPUT - Sample Application
  - INF: DBCS Support in Windows Versions 3.0 And 3.1
  - PRB: Accented Characters in Filename Controls Lose Accents
  - INF: Tips for Writing DBCS-Compatible Applications
  - INF: lstrcmpi, Accented Characters, and Sort Order in Windows
  - INF: Detailed Description of lstrcmp
  - INF: DDEEXEC.RTF - Technical Article

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

- INF: Menu Operations When MDI Child Maximized
  - INF: Modifying the System Menu of an MDI Child Window
  - INF: Terminating the Creation of an MDI Child Window
  - INF: Windows Does Not Support Nested MDI Client Windows
- PRB: Using Multiple Menus in an MDI Application
  - INF: Alternate MDI Tiling Scheme Code Sample Available
  - INF: Suggested Changes to Petzold's MDIDEMO Program
  - FIX: No Title for First MDI Child Window Icon
  - INF: Multiple Document Interface Enhancements in Windows 3.1
  - INF: WM_MDICREATE Message Documented Incorrectly
  - INF: Sample Code Demonstrates Superclassing MDI Client Window
- FIX: Setting the Focus to the Active MDI Child Window
  - FIX: Destroy Icon MDI Child Window, Title Remains
  - INF: Sample Code Demonstrates Window Status Bar
  - BUG: System Menu Wrong for CS_NOCLOSE-Style MDI Child
  - Using MoveWindow() to Move an Iconic MDI Child and Its Title
  - Creating a Hidden MDI Child Window

- INF: Minimal MDI Application Source in Software Library
- INF: Preventing an MDI Child Window from Changing Size
  - FIX: MDI Display Bad When Application Restored From Icon
  - FIX: Creating Zoomed MDI Child w/ Scroll Bar UAEs
  - PRWIN9012006: New Zoomed MDI Child Over Zoomed Child Flashes
  - FIX: Activate Next MDI Child, Present Child Zoomed
  - FIX: Windows Overrides MDI Child Window Styles
  - FIX: Double-Click Zoomed MDI Child System Menu NOP
- FIX: MDI Child Windows Ignore CS_NOCLOSE Style
  - BUG: MDI More Windows Dialog Activates Wrong Child
  - PRB: MDI Program Menu Items Changed Unexpectedly
  - INF: Changing the Default Background Color of an MDI Client
  - PRB: No Scroll Bars Displayed in an MDI Child Window
  - FIX: Maximized MDI Child with CS_NOCLOSE Problems
  - PRB: Processing the WM_QUERYOPEN Message in an MDI Application
- PRB: Pressing the ENTER Key in an MDI Application
  - SAMPLE: Customizing the MDI Window Menu
  - BUG: Iconic MDI Application Titles Do Not Update Properly

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

- INF: Inserting Right Justified Text in a Menu in Windows 3.0
  - INF: Adding Items to the System Menus of All Applications
  - INF: Customizing a Pop-Up Menu
  - INF: Changing How Pop-Up Menus Respond to Mouse Actions
- INF: How to Create a Menu-Bar Item Dynamically
  - PRB: MENUITEMTEMPLATE Structure Is Documented Incorrectly
  - INF: Sample Code Demonstrates Using TrackPopupMenu
  - INF: Switching Between Accelerator Tables in an Application
  - Owner-Draw Menu Item Width Includes a Check Mark Width
  - INF: Sample Code Demonstrates Adding Menus Dynamically
  - INF: Keeping Status Line Information About Menus Up-to-Date
  - INF: Managing Per-Window Accelerator Tables
- INF: Querying and Modifying the States of System Menu Items
  - INF:Win3.0 Top-Level Menu Items Unsupported in Owner-Draw Menu
  - PRB: SDK Sample Programs Define Delete Accelerator Incorrectly

- INF: Various Ways to Access Submenus and Menu Items
- INF: Appending Menu Items to Other Applications
- INF: Initializing Menus Dynamically
- PRB: FatalExit 0x0504 Produced from Incorrect lpTableName
- INF: GetMenuState() Can Return MF_BITMAP
- PRB: One Cause of RIP 0x0140 in USER When Accessing Menu
- PRB: TrackPopupMenu() on LoadMenuIndirect() Menu Causes UAE
- Reasons for Failure of Menu Functions
- INF: Creating Accelerator Tables Dynamically
- INF:Accessing Parent Window's Menu from Child Window w/ focus
- INF: Control Accelerators Conflict With Their ANSI Equivalents
- INF: Tracking Menu Selections in Windows Programs
- INF: Sample Code Implementing a Child Window with Menus
- INF: Mirroring Main Menu with TrackPopupMenu()

Messaging and Yielding

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

- INF: Using SendMessage() As Opposed to SendDlgItemMessage()
  - INF: Introduction to Nonpreemptive Multitasking in Windows
  - INF: GetInputState Is Faster Than GetMessage or PeekMessage
- INF: Using RegisterWindowMessage() to Communicate Between Apps
  - INF: How to Use PeekMessage Correctly in Windows
  - INF: Sample Code Demonstrates Background Processing
  - INF: Win 3.0 Icon WM_GETTEXT and WM_SETTEXT Docs Incomplete
  - INF: Differences Between PostAppMessage and PostMessage Funcs
  - INF: Background Processing with PeekMessage Code Example
  - INF: Performing Background Processing Without Using Timers
  - INF: When PostMessage() Will Return 0, Indicating Failure
- INF: Defining Private Messages for Application Use
  - INF: Some Basic Concepts of a Message-Passing Architecture

- INF:
- INF: Posting Frequent Messages Within an Application
- PRB: PeekMessage(hWnd==-1) Causes Fatal Exit 0x0007
- INF: Using PeekMessage() Loops in a Dialog Box
- INF: Handling WM_QUIT While Not in Primary GetMessage() Loop

INF: Nonzero Return from SendMsg() with HWND_BROADCAST

- INF: Message Retrieval in a DLL

Painting

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

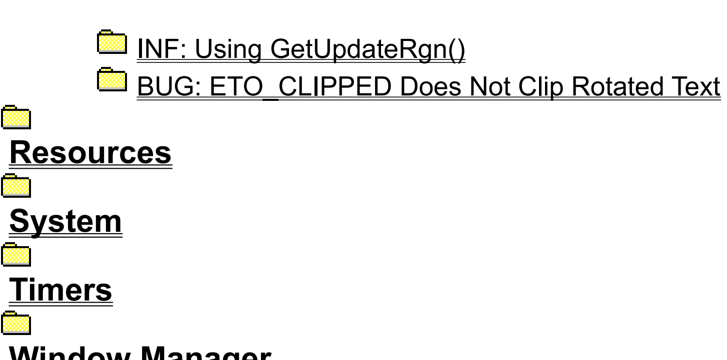
Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

- INF: Split Scrolling Using Two Windows
  - INF: CS_SAVEBITS Class Style Bit
  - INF: Using EndPaint() and BeginPaint()
- INF: Changing Window Colors with Control Panel
  - INF: Panning and Scrolling in Windows
  - INF: Using Blinking Text in an Application
  - INF: BeginPaint() Invalid Rectangle in Client Coordinates
  - INF: Using the WM_CTLCOLOR Message
  - INF: Designing Applications for High Screen Resolutions
  - FIX: TextOut Draws Dotted Underscore with Most Colored Text
- INF: How to Draw a Custom Window Caption
  - INF: GetClientRect() Coordinates Are Not Inclusive



INF: Using GetUpdateRgn()
BUG: ETO_CLIPPED Does Not Clip Rotated Text

Resources

System

Timers

Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

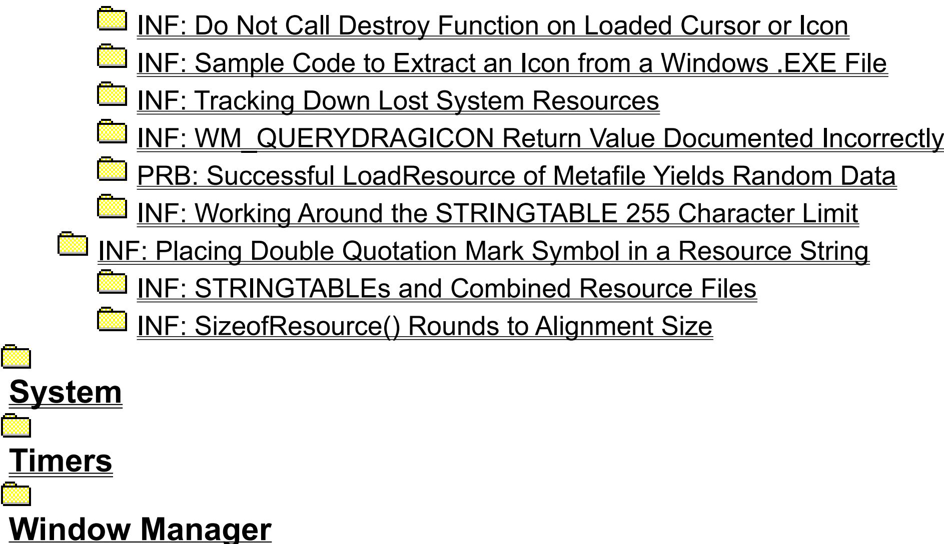
Menus and Accelerators

Messaging and Yielding

Painting

Resources

- INF: SetResourceHandler() Return Value for Callback Function
  - INF: FindResource() for Cursors and Menus
- INF: Save System Resources with One Control per Control Class
  - INF: Sample Code Extracts and Displays Application Resources
  - INF: Accessing Resources from a Windows Executable File
  - INF: Windows Resource Numbering Starts at 1
  - INF: Length of STRINGTABLE Resources
  - INF: Defining Format Resources and Binding Them to EXEs
  - INF: Sample Code Stores Resources in a Dynamic-Link Library
- INF: Multiple References to the Same Resource



- INF: Do Not Call Destroy Function on Loaded Cursor or Icon
- INF: Sample Code to Extract an Icon from a Windows .EXE File
- INF: Tracking Down Lost System Resources
- INF: WM_QUERYDRAGICON Return Value Documented Incorrectly
- PRB: Successful LoadResource of Metafile Yields Random Data
- INF: Working Around the STRINGTABLE 255 Character Limit
- INF: Placing Double Quotation Mark Symbol in a Resource String
  - INF: STRINGTABLEs and Combined Resource Files
  - INF: SizeofResource() Rounds to Alignment Size

System

Timers

Window Manager

Windows SDK Knowledge Base: User

- **Carets**
- **Classes**
- **Clipboard**
- **Controls**
- **Dynamic Data Exchange (DDE)**
- **Dynamic Data Exchange Management Library (DDEML)**
- **Dialog Boxes/Message Boxes**
- **Drag/Drop**
- **Extension Libraries**
- **File Formats**
- **Hooks**
- **INI Files**
- **Input - Mouse and Keyboard**
- **Localization/International**
- **Multiple Document Interface (MDI)**
- **Menus and Accelerators**
- **Messaging and Yielding**
- **Painting**
- **Resources**
- **System**
  - PRB: OpenFile Function Documented Incorrectly
  - INF: Possible Causes for System Resource Reduction
    - INF: Sample Code Demonstrates SystemParametersInfo
    - INF: GetCurrentTime() Rolls Over Every Hour
    - FIX: System Metrics Not Updated by SwapMouseButton()
    - INF: Sample Code to Provide Time and Date Information
    - INF: Calculating Free System Resources in Microsoft Windows
    - INF: Incomplete Description of ExitWindows()

Timers
Window Manager

Windows SDK Knowledge Base: User

Carets

Classes

Clipboard

Controls

Dynamic Data Exchange (DDE)

Dynamic Data Exchange Management Library (DDEML)

Dialog Boxes/Message Boxes

Drag/Drop

Extension Libraries

File Formats

Hooks

INI Files

Input - Mouse and Keyboard

Localization/International

Multiple Document Interface (MDI)

Menus and Accelerators

Messaging and Yielding

Painting

Resources

System

Timers

- INF: WM_TIMER Case on Page 94 of Guide to Programming Manual
- INF: GetCurrentTime and GetTickCount Functions Identical
  - INF: Keeping a Window on Top of All Other Windows
  - INF: General Information About Windows WM_TIMER Messages
  - FIX: Incorrect Parameter Sent to SetTimer Callback Function

Window Manager

Windows SDK Knowledge Base: User

- Carets
- Classes
- Clipboard
- Controls
- Dynamic Data Exchange (DDE)
- Dynamic Data Exchange Management Library (DDEML)
- Dialog Boxes/Message Boxes
- Drag/Drop
- Extension Libraries
- File Formats
- Hooks
- INI Files
- Input - Mouse and Keyboard
- Localization/International
- Multiple Document Interface (MDI)
- Menus and Accelerators
- Messaging and Yielding
- Painting
- Resources
- System
- Timers
- Window Manager
  - INF: How to Change a Window's Parent
    - INF: Determining Visible Window Area When Windows Overlap
    - INF: Process WM_GETMINMAXINFO to Constrain Window Size
    - INF: Determining the Topmost Pop-Up Window

  - INF: Ending the Windows Session from an Application
  - INF: Reasons Why RegisterClass() and CreateWindow() Fail
  - INF: Zooming Other Applications in Windows
- INF: Placing a Status Bar in an MDI Frame Window
  - INF: Subclassing the Desktop and Windows of Other Applications
  - INF: WindowFromPoint() Caveats
  - INF: Keeping a Window Iconic
  - PRB: WM_PARENTNOTIFY Message lParam Documented Incorrectly
  - INF: Obtaining an Application's Instance Handle
  - INF: Window Handles of Global Objects
- PRB: DeferWindowPos Function Documented Incorrectly
  - INF: Changing the Name of a Second Instance of an Application
  - PRB: Return from EnableWindow() Documented Incorrectly
  - INF: Preventing Windows from Switching Tasks
  - INF: Sample Code Simulates Changing List Box Style
  - PRB: Exchanging Window Handles Between 2 Cooperating Programs
  - INF: Translating Client Coordinates to Screen Coordinates
- INF: How to Create a Topmost Window
  - INF: Active Application, Active Window, Input Focus Definition
  - INF: Creating Applications that Task Manager Does Not Tile
  - PRB: Documentation Errors for DeferWindowPos and SetWindowPos
  - INF: Save and Restore Window Size, Position, Sample Code
  - INF: Cases Where
  - INF: Using the GetWindow() Function
- INF: Sample Code Demonstrates Windows 3.1 Window Styles
  - INF: Adding and Removing Caption of a Window
  - PRB: Window Dragged Close to Screen Edge Becomes Invisible
  - INF: Window Owners and Parents
  - INF: Allocating and Using Class and Window Extra Bytes
  - INF: Dangers of Uninitialized Data Structures
  - INF: Using SetClassLong Function to Subclass a Window Class
- PRB: DEF File Exports Statement Documentation Error
  - INF: WinQuickSort(), qsort() Replacement for Windows Available
  - INF: Reactivating First (and Only) Instance of an Application
  - INF: Sample Application Splits a Window into Two Panes
  - INF: Cannot Destroy Default Windows Help Menus and Buttons
  - INF: Algorithm Creates Window Same Size As Full-Screen Window
- INF: Guidelines for Allocating Instance (Per-Window) Data
  - INF: Using the DeferWindowPos Family of Functions
  - INF: Minimizing the MS-DOS Executive Window
  - INF: Both Windows in SetParent() Call Must Belong to Same Task
  - PRB: GetWindowPlacement Function Always Returns an Error
  - INF: Changing the Parent of a Child Window Using SetParent
  - INF: Transparent Windows
- INF: Different Ways to Close an Application Under Windows

INF: Adding to or Removing Windows from the Task List
INF: Safe Subclassing

PRB: Dialog Procedure Cannot Change Cursor of Control
Article ID:

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS
An application uses the method described in the documentation for WM_SETCURSOR in version 3.0 of the "Microsoft Windows Software Development Kit Reference Volume 1" on pages 6-98 and 6-99 to change the cursor for one of the child window controls in a dialog box, and the cursor does not change.

CAUSE
A dialog procedure cannot change the value returned by DefDlgProc() in response to WM_SETCURSOR.

RESOLUTION
Subclass the child window control in the dialog box and change the cursor in the subclass procedure.

More Information:

In the documentation for WM_SETCURSOR, the comments section states that when the message is not processed by a window, DefWindowProc() sends the message to the parent window to allow it to change the cursor. If the parent window's window procedure returns TRUE, further processing of the message is halted and the cursor specified by the parent window is used. This method gives the parent window control over the cursor used in a child window.

This technique does not work when a standard dialog box attempts to set the cursor for one of its child window controls because there is no method to change the value returned by DefDlgProc() in response to the WM_SETCURSOR message. DefDlgProc() is the window procedure for the standard dialog-box class. It calls the user-defined dialog procedure. If the dialog procedure returns FALSE to indicate that it did not process a message, DefDlgProc() performs the default processing.

The Boolean value returned from the dialog procedure in response to WM_SETCURSOR indicates only whether the dialog procedure has processed the message or not. It does not specify the value that DefDlgProc() should return in response to the message. [Note that the WM_CTLCOLOR, WM_COMPAREITEM, WM_VKEYTOITEM, and WM_CHARTOITEM messages are treated differently. The value returned by the dialog procedure in response to these messages is returned by DefDlgProc().]

Because it is not possible for the dialog procedure to specify the value returned by DefDlgProc() when it processes the WM_SETCURSOR message, it cannot indicate that a new cursor was specified for a

child window control. DefDlgProc() always returns FALSE in response to the WM_SETCURSOR message, regardless of how the dialog procedure processed the message. Therefore, the control's default cursor is always used because its parent window did not return TRUE in response to the WM_SETCURSOR message.

One way to work around this problem is to subclass the control that will have its cursor changed. The following code demonstrates how the subclass procedure can process the WM_SETCURSOR message:

```
   case WM_SETCURSOR:
       SetCursor(hCursor);
       return 0L;         // Do not pass this message on
```

This technique succeeds because the control sets its own cursor and does not pass the WM_SETCURSOR message to DefDlgProc().

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCrtCreate

INF: Controlling the Caret Color

Article ID: Q84054

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

When an application creates a custom caret using a bitmap, it is possible to specify white or black for the caret color. In the case of Windows running on a monochrome display, the application can cause the caret to be the color of the display (white, green, amber, and so forth, as appropriate). However, because the caret color is determined by Windows at run time based on the hardware installed, the application cannot guarantee what color will be used under all circumstances. This article provides information about using color in a custom caret.

More Information:

To create a caret, first create a bitmap with the desired pattern. To display the caret, Windows exclusive-ORs (XORs) and takes the opposite of the result (the NOT of the result) of the bitmap with the background of the client window. Therefore, to create a white caret, create a bitmap that when XOR'd with the window background will have an opposite value that will create a white color. It is when the caret blinks that Windows uses the reverse of the bitmap XOR'd with the background to draw the caret; this creates the white blink seen on the screen.

The bitmap for the caret cannot use a color palette. Windows does not use the color values from the palette in its calculations but the indices into the palette. While it is possible to use palette indices successfully, perfect symmetry of the colors in the palette is required. This is unlikely. For each color in the palette, its exact opposite color must be in the palette, in the exactly opposite index position.

However, when the application creates bitmaps itself, it has complete control over the bits. Therefore, the application can create the perfect counterpart that corresponds to the window background color. If the application uses this information to create the caret bitmap, when Windows creates the caret, it can choose the closest color available in the system palette.

Therefore, to create a white caret (or a black one, if the screen has too many light elements), the task is straightforward. Windows always reserves a few colors in the system palette and makes them available to all applications. On a color display, these colors include black and white. On a monochrome display, these colors are whatever the monochrome color elements are.

Because black and white (or the monochrome screen colors) are always available, the application simply creates a bitmap that, when XOR'd with the screen background color, produces black or white.
The technique involves one main principle: background XOR background = FALSE. Anything XOR'd with itself returns FALSE, which in bitmap terms maps to the color black.

The process of creating a caret from the background color involves the four steps discussed below:

1. Create a pattern brush the same as the window background

2. Select the pattern brush into a memory display context (DC)

3. Use the PATCOPY option of the PatBlt function to copy the brush pattern into the caret bitmap.

4. Specify the caret bitmap in a call to the CreateCaret function.

When this caret is XOR'd with the background, black will result. When the caret blinks, and is therefore displayed, Windows computes the opposite of the caret and XOR's this value into the background. This yields NOT(background XOR background) = NOT(FALSE) = TRUE which corresponds to WHITE. The first background represents the caret bitmap and the second is the current background color of the window.

Note that half the time the custom bitmap is displayed (when the caret "blinks") the other half of the time the background is displayed, (between "blinks").

If the background color is light gray or lighter [RGB values (128, 128, 128) through (255, 255, 255)], then a black caret is usually desired. The process of creating a black caret is just as straightforward. Modify step 1 of the process given above to substitute the inverse of the background for the background bitmap. When the caret blinks, it will show black. The equation that corresponds to this case is NOT(inverse of background XOR background) = NOT(TRUE) = FALSE which corresponds to BLACK.

To change the caret color to something other than black or white requires considerably more work, with much less reliable results because the application must solve the following equation:

```
   NOT(caret XOR background) = desired_color on the
                               "blink" of the caret.
```

where the value for the caret color must be determined given the desired color. A series of raster operations is required to solve this type of equation. (For more information on raster operations, see chapter 11 of the "Microsoft Windows [3.0] Software Development Kit Reference, Volume 2" or pages 573-585 of the "Microsoft Windows [3.1] Software Development Kit Programmer's Reference, Volume 3: Messages, Structures, and Macros.")

Even after solving this equation, the color actually displayed is controlled by Windows and the colors in the current system palette. With colors other than black or white, an exact match for the desired

color may not be available. In that case, Windows will provide the closest match possible. Because the palette is a dynamic entity and can be modified at will, it is impossible to guarantee a particular result color in all cases. The colors black and white should be safe most of the time because it is quite unusual for an application to modify the reserved system colors. Even when an application does change the system palette, it most likely retains a true black and a true white.

As long as a black and white remain in the palette (which is usually the case), this algorithm will provide a white or black caret.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCrtCreate

INF: Creating and Using a Custom Caret

Article ID: Q74514

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the Microsoft Windows graphical environment, creating a custom caret is simple. Windows has a series of caret control, creation, and deletion functions specifically designed to make manipulating the caret easy.

More Information:

The caret is a shared system resource. Unlike brushes, pens, device contexts and such, but like the cursor, only one caret is available under Windows. Also, like the cursor, an application can define a custom shape for the caret.

The CreateCaret function creates a custom caret. Its syntax is as follows:

```
   void CreateCaret(HWND hWnd, HBITMAP hBitmap,
                    int nWidth, int nHeight);
```

The caret shape can be a line, a block or, a bitmap specified as the hBitmap parameter. If the hBitmap parameter contains a valid handle (a bitmap handle returned from the CreateBitmap, CreateDIBitmap, or LoadBitmap functions), CreateCaret ignores the values of its nWidth and nHeight parameters and uses the dimensions of the bitmap. If hBitmap is NULL, the caret is a solid block; if hBitmap is one, the caret is a gray block. The nWidth and nHeight parameters specify the caret size in logical units. If either nWidth or nHeight is zero, the caret width or height is set to the window-border width or height.

If an application uses a bitmap for the caret shape, the caret can be in color; unlike the cursor, the caret is not restricted to monochrome.

CreateCaret automatically destroys the previous caret shape, if any, regardless of which window owns the caret. The new caret is initially hidden; call the ShowCaret function to display the caret.

Because the caret is a shared resource, a window should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive. Only the window that owns the caret should move it, show it, hide it, or modify it in any way.

Other functions related to the caret are the following:

- SetCaretPos

This function moves the caret to the specified position (in logical coordinates).

- GetCaretPos
  This function retrieves the caret's current position (in screen coordinates).

- ShowCaret
  This function shows the caret on the display at the caret's current position. When shown, the caret flashes automatically. If the caret is not owned by the window specified in the call, the caret is not shown.

- HideCaret
  This function hides the caret by removing it from the display screen. HideCaret hides the caret only if the window handle specified in the call is the window that owns the caret. Hiding the caret does not destroy it.

  Note: Hiding the caret is cumulative; ShowCaret must be called once for every call to HideCaret. For example, if HideCaret is called five times, ShowCaret must be called five times for the caret to be shown.

- DestroyCaret
  This function removes the caret from the screen, frees the caret from the current owner-window, and destroys the current shape of the caret. It destroys the caret only if the current task owns the caret. This call should be used in conjunction with CreateCaret. DestroyCaret does not free or destroy a bitmap used to define the caret shape.

- SetCaretBlinkTime
  This function sets the caret blink rate. After the blink rate is set, it remains the same until the same window changes it again, another window changes it, another application changes it, or Windows is rebooted.

- GetCaretBlinkTime
  This function returns the current caret blink rate.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCrtCreate

INF: Creating a Nonblinking Caret

Article ID: Q74607

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Microsoft Windows graphical environment is designed to provide a blinking caret. However, using a timer and the SetCaretBlinkTime function, an application can "trick" the caret into not blinking.

More Information:

Although Windows is designed to blink the caret at a specified interval, a timer function and SetCaretBlinkTime can be used to prevent Windows from turning the caret off by following these three steps:

1. Call SetCaretBlinkTime(10000), which instructs Windows to blink the caret every 10,000 milliseconds (10 seconds). This results in a "round-trip" time of 20 seconds to go from OFF to ON and back to OFF (or vice versa).

2. Create a timer, using SetTimer, specifying a timer procedure and a 5,000 millisecond interval between timer ticks.

3. In the timer procedure, call SetCaretBlinkTime(10000). This resets the timer in Windows that controls the caret blink.

When an application implements this procedure, Windows never removes the caret from the screen, and the caret does not blink.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCrtBlinkrate

INF: Using Extra Fields in Window Class Structure

Article ID: Q10841

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In order to generate several child windows of the same class, each having its own set of static variables and independent of the sets of the variables in the sibling windows, you need to use the cbWndExtra field in WNDCLASS, the window-class data structure, when registering a window; then, use SetWindowWord() (or Long) and GetWindowWord() (or Long). These functions will either get or set additional information about the window identified by hWnd.

Use positive offsets as indexes to access any additional bytes that were allocated when the window class structure was created, starting at zero for the first byte of the extra space. Similarly, if you want to refer to bytes already defined by Windows within the structure, use offsets defined with the GWW and GWL prefixes.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrClsGetclswrd

INF: Use of Allocations w/ cbClsExtra & cbWndExtra in Windows
Article ID: Q11606

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following is an explanation of the use of the allocations with cbClsExtra and cbWndExtra?:

1. cbClsExtra -- extra bytes to allocate to CLASS data structure in USER.EXE local heap when RegisterClass() is called. Accessed by Get/Set CLASS Word/Long ();.

2. cbWndExtra -- extra bytes to allocate to WND data structure in USER.EXE local heap when CreateWindow() is called. Accessed by Get/Set WND Word/Long ();.

You can use these structures at your discretion and for any purpose you desire.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrClsRareMisc

INF: Scope of Window Classes

Article ID: Q28353

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

This article discusses the scope of window classes created by the RegisterClass function and used by the CreateWindow function.

When an application calls the RegisterClass function, Windows determines if that application has already registered a class with the same name. If it has, registration fails because a module can register only one class under a given name. When an application calls the CreateWindow function to create a window of a specified class, Windows looks in the list of registered classes for a class with the specified class name registered by the calling module. If the class is not found, Windows looks in the classes list for the specified class name registered by any module. If the class name is not found, the call to CreateWindow fails.

This implies that a module can create a window using any other module's class if and only if it has not registered a class with the same name. If a module registers a class with the same name as another module's class, the module can use only its own version. However, consider the following situation regarding the MyClass window class:

- Application A and Application B each register window classes named MyClass.

- Application C, which has not registered a MyClass window class, calls CreateWindow to create a MyClass-class window.

One cannot predict whether Windows will create the window using Application A's MyClass class or Application B's MyClass class. To be safe, the name of each window class should include the registering application's name.

An application can reference a class defined by a dynamic-link library (DLL). Under versions of Windows earlier than 3.1, an application should not reference window classes registered by other applications. This restriction is caused by the expanded memory manager (EMM) provided by Windows real mode. Under certain conditions, the window procedure in the module that registers a class is not available while the application that created the window is running.

Additional reference words: 2.03 2.10 3.00 3.10 2.x TAR73212
KBCategory:
KBSubcategory: UsrClsReg

PRB: SetClipboardData Function Documentation Incomplete

Article ID: Q71413

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

The documentation for the SetClipboardData function on pages 4-369 through 4-371 of the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0, and on pages 826-828 of the "Microsoft Windows Software Development Kit Version 3.1 Programmer's Reference, Volume 2: Functions" is incomplete.

RESOLUTION

The following information should be added to the "Return Value" section on page 4-370 of the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0, and on page 826 of the "Microsoft Windows Software Development Kit Version 3.1 Programmer's Reference, Volume 2: Functions" manual:

If NULL is specified as the hMem parameter to SetClipboardData (which signifies delayed clipboard rendering), SetClipboardData will return NULL.

Additional reference words: 3.00 3.10 3.x docerr MICS3 R1.12
KBCategory:
KBSubcategory: UsrClp

INF: Clipboard Memory Sharing in Windows

Article ID: Q11654

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following are questions about Clipboard memory sharing:

1. Q. Does the Clipboard UNLOCK before freeing the handle when I tell it to SetClipboardData()?

   A. Yes, the Clipboard UNLOCKs before freeing the handle when you SetClipboardData().

2. Q. Does the Clipboard actually copy my global storage to another block, or does it just retain the value of my handle for referencing my block?

   A. The Clipboard is sharable; it retains the value of the handle.

3. Q. Does GetClipboardData() remove the data from the Clipboard, or does it allow me to reference the data without removing it from the Clipboard?

   A. The data handle returned by GetClipboardData() is controlled by the Clipboard, not by the application. The application should copy the data immediately, instead of relying on the data handle for long-term use. The application should not free the data handle or leave it locked. To remove data from the Clipboard, call SetClipboardData().

Additional reference words: 2.x 3.00
KBCategory:
KBSubcategory: UsrClpCopy

INF: SetClipboardData() and CF_PRIVATEFIRST
Article ID: Q24252

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Problem:

The documentation for SetClipboardData() states that CF_PRIVATEFIRST can be used to put private data formats on the clipboard. It also states that data of this format is not automatically deleted. However, that is apparently not true. That is, the data is removed automatically when the clipboard is emptied (sending a WM_DESTROYCLIPBOARD message) and the new item is set into the clipboard. The old item is shown; however, the handle now is invalid because GlobalFree() is called on it.

Response:

GlobalFree() was not called on this handle. If you try to use the other handle to this memory, you will find that the one you initially received from GlobalAlloc() is still valid. Only the clipboard handle has been invalidated by the call to EmptyClipboard().

The documentation states that "Data handles associated (with CF_PRIVATEFIRST) will not be freed automatically." This statement refers to the memory associated with that data handle. When SetClipboardData() is called under standard data types, it frees the block of memory identified by hMem. This is not the case for CF_PRIVATEFIRST. Applications that post CF_PRIVATEFIRST items on the clipboard are responsible for the memory block containing those items.

This is not intended to imply that items placed on the clipboard will remain on the clipboard if they are CF_PRIVATEFIRST. When a call is made to EmptyClipboard(), all objects will be removed.

Additional reference words: 2.00 3.00
KBCategory:
KBSubcategory: UsrClpFormats

INF: Return Value from ChangeClipboardChain()

Article ID: Q28338

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The return value for ChangeClipboardChain() is TRUE if hWnd (the value to be removed) is the last one on the chain. Otherwise, the value is the return value [via SendMessage()] from the final item in the Clipboard chain, which should be TRUE. The change can successfully be made, however, even if the final window in the chain returns FALSE. This merely indicates that the application with the final window in the chain is handling the message incorrectly.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrClpRareMisc

INF: The Clipboard and the WM_RENDERFORMAT Message
Article ID: Q31668

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The clipboard sends a WM_RENDERFORMAT message to an application to request that application format the data last copied to the clipboard in the specified format, and then pass a handle to the formatted data to the clipboard.

If an application cannot supply the requested data, it should return a NULL handle. Because most applications provide access to the actual data (not rendered) through the CF_TEXT format, applications that use the clipboard can get the applicable data when rendering fails.

If the application cannot render the data because the system is out of memory, the application can call GlobalCompact(-1) to discard and compress memory, then try the memory allocation request again.

If this fails to provide enough memory, the application can render the data into a file. However, applications that use this technique must cooperate in order to know that the information is in a file, the name of the file, and the format of the data.

Additional reference words: 2.x 3.00
KBCategory:
KBSubcategory: UsrClpFormats

INF: WM_SIZECLIPBOARD Must Be Sent by Clipboard Viewer App

Article ID: Q74274

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

The WM_SIZECLIPBOARD message is not generated by the Windows graphical environment; the clipboard viewer generates this message.

An application that can display the clipboard contents, such as the CLIPBRD.EXE application in Windows version 3.0, is a clipboard viewer. When the size of the clipboard viewer's client area changes and the clipboard contains a data handle for the CF_OWNERDISPLAY format, the clipboard viewer must send the WM_SIZECLIPBOARD message to the current clipboard owner. The GetClipboardOwner function returns the window handle of the current clipboard owner. This window handle is the handle in the last call to OpenClipboard.

When sending the WM_SIZECLIPBOARD message, the clipboard viewer must specify two parameters. The wParam parameter identifies the clipboard viewer's window handle. The low-order word of the lParam parameter contains the handle to a block of global memory that holds a RECT data structure. The RECT structure defines the area in the clipboard viewer that the clipboard owner should paint.

Additional reference words: 2.03 2.10 3.00 3.10 2.x
KBCategory:
KBSubcategory: UsrClpView

PRB: Private Data Formats Freed by the Clipboard
Article ID: Q85430

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The documentation for the SetClipboardData function on pages 826-828 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" manual states that data placed on the clipboard in a private data format is not automatically freed when removed from the clipboard. The identifiers for private data formats are in the range CF_PRIVATEFIRST through CF_PRIVATELAST. The statement in the manual is incorrect. Data that is placed on the clipboard in a private format is automatically freed when the data is removed from the clipboard.

More Information:

Clipboard data in a private data format is automatically freed when the data is removed from the clipboard. Windows sends the WM_DESTROYCLIPBOARD message to the application that put the data on the clipboard before the data is freed. An application can free any corresponding internal data when it receives the WM_DESTROYCLIPBOARD message.

Private data in the CF_OWNERDISPLAY format is not automatically freed when the data is removed from the clipboard. An application must free data of this format when the application receives a WM_DESTROYCLIPBOARD message.

Additional reference words: 3.00 3.10 docerr
KBCategory:
KBSubcategory: UsrClpFormats

INF: Method for Sending Text to the Clipboard
Article ID: Q35100

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Sending text to the Clipboard is usually a cumbersome process of allocating and locking global memory, copying the text to that memory, and sending the Clipboard the memory handle. This method involves many pointers and handles and makes the entire process difficult to use and understand.

Clipboard I/O is easily accomplished with an edit control. If a portion of text is highlighted, an application can send the edit control a WM_COPY or WM_CUT message to copy or cut the selected text to the Clipboard. In the same manner, text can be pasted from the Clipboard by sending a WM_PASTE message to an edit control.

The following example demonstrates how to use an edit control transparently within an application to simplify sending and retrieving text from the Clipboard. Note that this code will not be as fast as setting or getting the Clipboard data explicitly, but it is easier from a programming standpoint, especially if the text to be sent is already in an edit control. Note also that the presence of the edit window will occupy some additional memory.

More Information:

For simplified Clipboard I/O, do the following:

1. Declare a global HWND, hEdit, which will be the handle to the edit control.

2. In WinMain, use CreateWindow() to create a child window edit control. Use the style WS_CHILD, and give the control dimensions large enough to hold the most text that may be sent to or received from the Clipboard. CreateWindow() returns the handle to the edit control that should be saved in hEdit.

3. When a Cut or Copy command is invoked, use SetWindowText() to place the desired string in the edit control, then use SendMessage() to select the text and copy or cut it to the Clipboard.

4. When a Paste command is invoked, use SetWindowText() to clear the edit control, then use SendMessage() to paste text from the Clipboard. Finally, use GetWindowText() to copy the text in the edit control to a string buffer.

The actual coding for this procedure is as follows:

```
.
.
.

#define ID_ED    100
HWND       hEdit;

.
.
.
/* In WinMain: hWnd is assumed to be the handle of the parent window, */
/* hInstance is the instance handle of the parent.                    */
/* The "EDIT" class name is required for this method to work. ID_ED   */
/* is an ID number for the control, used by Get/SetDlgItemText.       */

hEdit=CreateWindow("EDIT",
                   NULL,
                   WS_CHILD | BS_LEFTTEXT,
                   10, 15, 270, 10,
                   hWnd,
                   ID_ED,
                   hInstance,
                   NULL);


.
.
.

/* In the procedure receiving CUT, COPY, and PASTE commands:         */
/* Note that the COPY and CUT cases perform the same actions, only   */
/* the CUT case clears out the edit control.                         */

/* Get the string length */
short    nNumChars=strlen(szText);

case CUT:
    /* First, set the text of the edit control to the desired string */
    SetWindowText(hEdit, szText);

    /* Send a message to the edit control to select the string */
    SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(0, nNumChars));

    /* Cut the selected text to the clipboard */
    SendMessage(hEdit, WM_CUT, 0, 0L);
    break;

case COPY:
    /* First, set the text of the edit control to the desired string */
    SetWindowText(hEdit, szText);

    /* Send a message to the edit control to select the string */
    SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(0, nNumChars));

    /* Copy the text to the clipboard */
    SendMessage(hEdit, WM_COPY, 0, 0L);
    break;
```

```
    case IDM_PASTE:
        /* Check if there is text available */
        if (IsClipboardFormatAvailable(CF_TEXT))
            {
            /* Clear the edit control */
            SetWindowText(hEdit, "\0");

            /* Paste the text in the clipboard to the edit control */
            SendMessage(hEdit, WM_PASTE, 0, 0L);

            /* Get the test from the edit control into a string.    */
            /* nNumChars represents the number of characters to get */
            /* from the edit control.                               */
            GetWindowText(hEdit, szText, nNumChars);
            }
        else
            MessageBeep(0); /* Beep on illegal request */
        break;
```

Additional reference words: 2.x 3.00
KBCategory:
KBSubcategory: UsrClpCopy

PRB: Avoid GDI Object Private Clipboard Formats

Article ID: Q89119

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In the Microsoft Windows Software Development Kit (SDK) version 3.1 "Programmer's Reference Volume 2: Functions," page 828 of the documentation for the SetClipboardData function states:

  Private data formats in the range CF_GDIOBJFIRST through CF_GDIOBJLAST will be automatically removed by a call to the DeleteObject function when the data is removed from the clipboard.

This statement is incorrect under Windows 3.1. Private data formats with types in this range are treated like global memory handles by Windows. As such, this range of private data formats should be avoided.

More Information:

Under the debugging version of Windows, if an application calls the SetClipboardData function with a handle to a GDI object, and a format identifier in the range CF_GDIOBJFIRST through CF_GDIOBJLAST, Windows will output the following message on the debugging terminal:

```
  err APPNAMEHERE->USER GLOBALREALLOC+15: Invalid global handle: 0x1234
```

This is due to a problem in Windows version 3.1. Windows treats the GDI object's handle as a global memory handle, when it should treat it as a GDI handle (for example, a handle to a brush, palette, font, and so on.)

In addition, the object will not be deleted properly when the EmptyClipboard function is called. This leads to a loss of memory in GDI's heap. When GDI's heap is exhausted, either Windows crashes or output is corrupted.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrClpFormats

INF: XLTABLE Clipboard Format Documentation Available

Article ID: Q75631

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

XLTABLE is a new clipboard format developed for use in dynamic-data exchange (DDE) conversations with Microsoft Excel.

Using this format instead of BIFF (the Binary Interchange File Format) results in faster conversations. Code to read and write this format is also easier to implement. The format does not introduce translation problems that can occur when putting text on the clipboard.

The "Microsoft Excel Software Development Kit," part number 065-000-020, describes all the Excel file formats. In the United States, this document is available for $20.00 through the Microsoft Consultant Relations Program by calling (800) 227-4679. Outside the United States, contact the Microsoft subsidiary that supports your country for pricing and availability information.

More Information:

The following information documents the XLTABLE data format:

FAST FILE FORMAT

The Fast DDE Table data consists of multiple simple_datablocks, one after the other:

```
   Fast_DDE_Table = <simple_datablock> <simple_datablock> ...
       <simple_datablock>
```

A simple_datablock has three parts:

```
    WORD tdt;       // The table datatype
    WORD cb;        // The size (in bytes) of the data part
    BYTE data[];    // There are cb bytes in this part
```

Description of the Table Datatypes:

```
tdt         Value    Meaning
---         -----    -------

tdtTable    0x0010   Gives the size of the table. The cb is 4.
                     The data (4 bytes) has two words: the first
                     word is the number of rows, the second word
                     is the number of columns.
```

tdtFloat	0x0001	The data contains floating point values in IEEE real format. The size of each entry is 8 bytes.
tdtString	0x0002	The data contains strings in "st" format. In this format, the first byte gives the length of the string, and the following bytes make up the string. The string is not null-terminated.
tdtBool	0x0003	Boolean values: 1 = TRUE, 0 = FALSE Each entry is 2 bytes.

tdtError	0x0004	Error values for the cell:
		0 = #NULL!
		7 = #DIV/0!
		15 = #VALUE!
		23 = #REF!
		29 = #NAME?
		36 = #NUM!
		42 = #N/A
		Each entry is 2 bytes.

tdtBlank	0x0005	The cb is 2. The data (2 bytes) has one word. The value of the word is the number of blank cells.
tdtType	0x0080	Not used. Gives the type of the cell. Each entry is 2 bytes.
tdtFormat	0x0081	Not used. Gives the format of the cell (0-relative index into the format table as shown in the Format Number dialog box). Each entry is 2 bytes.

Order of the cells: The first simple_datablock must be of type tdtTable. This record provides the number of rows and columns. This is followed by rows*cols cells represented by the subsequent simple_datablocks. The cells are represented row-wise, (all cells of the first row are listed first, then all cells of the second row, and so on).

For example, the Fast DDE Table for a selection consisting of three cells in a row, containing the strings "East", "West", and "North", respectively, will resemble the following:

```
   10 00 04 00 01 00 03 00    - tdtTable, 4, 1, 3
   02 00 10 00                - tdtString, 16
      04 45 61 73 74          - 4, East
      04 57 65 73 74          - 4, West
      05 4e 6f 72 74 68       - 5, North
```

Additional reference words: 3.00 3.10 DDE clipboard
KBCategory:
KBSubcategory: UsrClpFormats

PRB: WM_RENDERFORMAT Documentation Incomplete
Article ID: Q75975

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The documentation for the WM_RENDERFORMAT message on page 6-98 of the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0, and on page 188 of the "Microsoft Windows Programmer's Reference Volume 3: Messages, Structures, and Macros" manual for version 3.1 is incomplete.

The following information should be added to the documentation for the function as a Comments section:

> During the processing of this message, the OpenClipboard and CloseClipboard functions should NOT be called. The WM_RENDERFORMAT message is sent from within the GetClipboardData function which an application calls from within a block of code surrounded by the OpenClipboard and CloseClipboard functions. While processing the WM_RENDERFORMAT message, the application should not again call the OpenClipboard and CloseClipboard functions.

NOTE: The first sentence of the above information is already included on page 188 of the "Microsoft Windows Programmer's Reference Volume 3: Messages, Structures, and Macros" manual for version 3.1.

Note that while processing the WM_RENDERALLFORMATS message, the OpenClipboard and CloseClipboard functions must be called in order to call the SetClipboardData function. For more information, query on the following words:

   prod(winsdk) and WM_RENDERALLFORMATS and OpenClipboard

Additional reference words: 3.00 3.10 3.x docerr MICS3 R1.12 R5.5
KBCategory:
KBSubcategory: UsrClpFormats

INF: Reasons for Failure of Clipboard Functions

Article ID: Q92530

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

The following clipboard functions:

OpenClipboard()
CloseClipboard()
EmptyClipboard()
GetClipboardData()
SetClipboardData()
EnumClipboardFormats()
SetClipboardViewer()
ChangeClipboardChain()
GetOpenClipboardWindow()
GetClipboardOwner()

can fail for several reasons. Different functions return different values to indicate failure. Read the documentation for information about each function. This article combines the causes of failure for all functions and provides a resolution or explanation. In the More Information section, a list of affected functions follows each cause. The causes are:

1. The clipboard is not opened by any application.

2. The current application does not have the clipboard open.

3. The current application does not own the clipboard.

4. User's data segment is full.

5. Insufficient global memory.

6. The specified clipboard format is not supported.

7. The application that set the clipboard data placed a corrupt or invalid metafile in the clipboard.

8. An application is attempting to open an already open clipboard. The debug mode of Windows 3.1 will send the "Clipboard already open" message.

9. The application that opened the clipboard used NULL as the window handle.

More Information:

Cause 1: The clipboard is not opened by any application.

Resolution 1: Open the clipboard using OpenClipboard(). If a DLL needs to open the clipboard, it may pass hwnd = NULL to OpenClipboard().

Explanation 1: An application cannot copy data (using SetClipboardData()) when no application has the clipboard currently open.

Affected Functions: SetClipboardData().

Cause 2: The current application does not have the clipboard open.

Resolution 2: Open the clipboard using OpenClipboard(). If a DLL needs to open the clipboard, it may pass hwnd = NULL to OpenClipboard().

Explanation 2: An application cannot empty or close the clipboard without first opening it.

Affected Functions: EmptyClipboard(), CloseClipboard().

Cause 3: The current application does not own the clipboard.

Resolution 3: Open the clipboard and get ownership by emptying it.

Explanation 3: An application cannot enumerate the clipboard formats without owning it.

Affected Functions: EnumClipboardFormats().

Cause 4: User's data segment is full.

Explanation 4: There should be space available in User's data segment to store internal data structures when SetClipboardData() is called.

Affected Function: SetClipboardData().

Cause 5: Insufficient global memory.

Explanation 5: If the clipboard has data in either the CF_TEXT or CF_OEMTEXT format and if GetClipboardData() requests text in the unavailable format, then Windows will perform the conversion. The converted text must be stored in global memory.

Affected Function: GetClipboardData().

Cause 6: The specified clipboard format is not supported.

Resolution 6: Use IsClipboardFormatAvailable() to check whether the specified format is available on the clipboard.

Affected Function: GetClipboardData().

Cause 7: The application that set the clipboard data placed a corrupt or invalid metafile in the clipboard.

Resolution 7: There are no functions to tell whether a given metafile is corrupt or invalid. Try playing the metafile and see if the metafile plays as expected.

Affected Function: SetClipboardData().

Cause 8: Application is attempting to open an already open clipboard. The debug mode of Windows 3.1 will send the "Clipboard already open" message.

Explanation 8: The clipboard must be closed by the application that opened it, before other applications can open it.

Affected Functions: OpenClipboard().

Cause 9: The application that opened the clipboard used NULL as the window handle.

Explanation 9: An application can call OpenClipboard(NULL) to successfully open a clipboard. The side effects are that subsequent calls to GetClipboardOwner() and GetOpenClipboardWindow() return NULL. An application can also call SetClipboardViewer(NULL) successfully. However, there is no reason why this should be allowed, and it is currently reported as a bug. The side effects are that subsequent calls to GetClipboardViewer() and ChangeClipboardChain() return NULL. NULL from these functions does not necessarily imply that they failed.

Affected Functions: GetClipboardOwner(), GetOpenClipboardWindow(), GetClipboardViewer(), ChangeClipboardChain().

Additional reference words: 3.00 3.10 fails GetClipboardFormatName RegisterClipboardFormat
KBCategory:
KBSubcategory: UsrClp

PRB: WM_RENDERALLFORMATS Documentation Incomplete
Article ID: Q75976

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The documentation for the WM_RENDERALLFORMATS message on page 6-98 of the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0 is incomplete.

The following information should be added to the documentation in the Comments section:

   During the processing of this message, the OpenClipboard and CloseClipboard functions must be called in order for the application to call the SetClipboardData function.

This additional information is included on Page 187 of the "Microsoft Windows Programmer's Reference Volume 3: Messages, Structures, and Macros" manual for version 3.1.

Note that in processing the WM_RENDERFORMAT message, an application should NOT call the OpenClipboard and CloseClipboard functions. For more information query on the following words:

   prod(winsdk) and WM_RENDERFORMAT and OpenClipboard

Additional reference words: docerr MICS3 R1.12 R5.5
KBCategory:
KBSubcategory: UsrClpFormats

INF: WM_SIZECLIPBOARD Message Documented Incorrectly

Article ID: Q77360

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Page 6-103 of the "Microsoft Windows Software Development Kit Reference Volume 1" incorrectly describes the contents of the lParam parameter. The correct documentation is as follows:

> The low-order word of the lParam parameter contains a global handle to a RECT data structure that specifies the area the clipboard owner should paint. The high-order word is not used.

In the comments section, modify the reference to a PAINTSTRUCT structure in the second paragraph to refer to a RECT.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrClp

BUG: WM_CTLCOLOR Message Not Sent to Combo Box
Article ID: Q75939

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9109001

SYMPTOMS
   When a combo box window is subclassed to process the WM_CTLCOLOR
   message to change the color of its edit control and list box child
   windows, the child windows do not change color.

CAUSE
   The WM_CTLCOLOR message is not sent to the combo box window.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. We are researching this problem and will post new information
   here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Placing a Caret After Edit-Control Text

Article ID: Q12190

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The EM_SETSEL message can be used to place a selected range of text in a Windows edit control. If the starting and ending positions of the range are set to the same position, no selection is made and a caret can be placed at that position. To place a caret at the end of the text in a Windows edit control and set the focus to the edit control, do the following:

```
hEdit = GetDlgItem( hDlg, ID_EDIT );    // Get handle to control
SetFocus( hEdit );
SendMessage( hEdit, EM_SETSEL, 0, MAKELONG(0xffff,0xffff) );
```

It is also possible to force the caret to a desired position within the edit control. The following code fragment shows how to place the caret just to the right of the Nth character:

```
hEdit = GetDlgItem( hDlg, ID_EDIT );    // Get handle to control
SetFocus( hEdit );
SendMessage( hEdit, EM_SETSEL, 0, MAKELONG(N,N) );
                                       // N is the character position
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Placing Captions on Control Windows

Article ID: Q77750

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The SetWindowText() function can be used to place text into the
caption bar specified for a control window. The control must have the
WS_CAPTION style for the caption to be visible.

This technique does not work with edit controls because the
SetWindowText() function specifies the contents of the edit control,
not its caption.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtl

INF: Right Justifying Numbers in a Windows 3.0 List Box

Article ID: Q64078

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In versions of Windows earlier than 3.0, it is possible to align columns of information in a list box with spaces because all characters in the system font are the same width. In Windows 3.0, the system font is proportionally spaced with letters of differing widths, which prevents this technique from being usable.

Tabs can be used to align columns in a list box. The LBS_USETABSTOPS style must be specified when the list box is created. Then, set a tab to correspond with the position of each character of each numeric column. The tabs are specified in dialog units, which are one-fourth of a character width. For example, a three-digit field that should align 12 characters from the left edge of the dialog box can be implemented by setting tabs at positions 40, 44, and 48. This is illustrated below:

```
+------------+
|text     xxx|
|longer    xx|
          ^^^
          ||+- position 48
          |+-- position 44
          +--- position 40
```

After determining the text to insert in the list box, convert the spaces to the appropriate number of tabs as the following code fragment demonstrates:

```
sprintf(szNumBuffer, "%3d", Number);
for (i = 0; i < 3; i++)
  if (szNumBuffer[i] == ' ')
     szNumBuffer[i] = '\t';
```

This code uses the three tabs set above.

There is a file in the Software/Data Library named RIGHTJUS that illustrates this convention. RIGHTJUS can be found in the Software/Data Library by searching on the word RIGHTJUS, the Q number of this article, or S12663.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Creating a List Box Without a Scroll Bar

Article ID: Q11365

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When LBS_STANDARD is used as follows

```
   LBS_NOTIFY | LBS_SORT | WS_BORDER | LBS_STANDARD
```

the following results (as defined in WINDOWS.H):

```
   LBS_STANDARD = #00A00003;
                 /* LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER */
```

To create a dialog box that contains a list box without the vertical scroll bar, use NOT WS_VSCROLL as the style for creating a list box control without a vertical scroll bar, as follows:

```
   (LBS_STANDARD & ~WS_VSCROLL)  // NOT WS_VSCROLL
```


Additional reference words: 2.00 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Retrieve Line Number from Edit Control Sample Code
Article ID: Q77782

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In an edit control, the EM_LINEFROMCHAR message returns the line number of the line that contains a specified character.

EDITCNT is a file in the Software/Data Library that demonstrates using the EM_LINEFROMCHAR message to determine the line number of the line of an edit control selected using the mouse.

EDITCNT can be found in the Software/Data Library by searching on the word EDITCNT, the Q number of this article, or S12255. EDITCNT was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Owner-Draw: 3-D Push Button Made from Bitmaps with Text

Article ID: Q64328

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

An owner-draw button can be defined and drawn to provide any desired appearance, even to mimic the three-dimensional push buttons of Windows version 3.0. With an owner-draw button, the application can control the button face color, unlike a standard push button, which always has a gray face.

More Information:

ODBUTTON is a file in the Software/Data Library that contains an example of an owner-draw button. The button uses two 64-by-64-bit bitmaps. The first bitmap represents the default button state. In this state, the button has a light gray face, a light left and upper border, and the button's text is centered in the bitmap. The second bitmap represents the pressed button state. In this state, the button has a solid, dark gray face, and the button's text is shifted to the right and down two pixels.

ODBUTTON can be found in the Software/Data Library by searching on the word ODBUTTON, the Q number of this article, or S12669. ODBUTTON was archived using the PKware file-compression utility.

The ODBUTTON sample source code can be used as a template for creating other applications. The main task involves creating appropriate bitmaps.

Under Windows 2.x, an application can change the color of the button face by processing WM_CTLCOLOR messages. This functionality is not available in Windows 3.0. However, the more powerful owner-draw feature can be used to achieve the same effect.

Like a Windows 3.0 standard push button, this sample owner-draw button indicates that it has the focus by drawing a dashed, black rectangular border around the button text. This is accomplished by using the new Windows 3.0 function, DrawFocusRect(). Like the text of the button, the focus rectangle must also be shifted right and down two pixels when the button is pressed.

Bitmaps are created in a fixed size, in pixels, regardless of the display resolution used. An application can determine which display is in use by reading the SYSTEM.INI file. Two alternatives are available to create buttons that are an appropriate size on different displays, as follows:

1. Create different pairs of default/pressed bitmaps for each display

resolution the application is targeted (for example, Hercules monochrome, CGA, EGA, VGA, super VGA, and 8514).

2. Create separate bitmaps for the borders of the button and for the button faces. During the initialization of the application, make four calls to StretchBlt(). The first call should modify the left and right borders vertically. The second call should change the top and bottom borders horizontally. The third and fourth calls should modify each button face bitmap both horizontally and vertically. Doing separate calls eliminates distortion to the border widths as a display-specific bitmap is created.

Apart from the information discussed above, owner-draw buttons are handled like other owner-draw controls. However, Windows does not send the application a WM_MEASUREITEM message to define the dimensions of the button control dynamically. The dimensions of the owner-draw button are specified in a dialog box template or CreateWindow() call, similar to the dimensions of nonowner-draw buttons.

Like other owner-draw controls, the application draws the owner-draw button in response to a WM_DRAWITEM message. Drawing should take into account the selection and focus states of the control, as well as the normal state without either selection or focus.

For more information on processing the WM_DRAWITEM message, query on the following words:

    prod(winsdk) and WM_DRAWITEM

For more information on owner-draw controls in general, query on the following words:

    prod(winsdk) and owner and draw

Additional reference words: 3.00 softlib ODBUTTON.ZIP
KBCategory:
KBSubcategory: UsrCtlOdbuttons

FIX: Problem Adding Horizontal Scroll Bar to List Box

Article ID: Q64502

```
----------------------------------------------------------------------
The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0
----------------------------------------------------------------------
```

```
Summary:

PROBLEM ID: WIN9012020

SYMPTOMS
   After a list box has been created, an LB_SETHORIZONTALEXTENT
   message will not cause a horizontal scroll bar to be added to or
   removed from the list box.

RESOLUTION
   Microsoft has confirmed this to be a problem in Windows version
   3.0. If the scroll bar will be required at any time, set a large
   scroll range with the LB_SETHORIZONTALEXTENT message at the time of
   initialization. The scroll bar can be disabled by sending another
   message with a small scroll range.

   This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox
```

INF: Multicolumn List Boxes in Microsoft Windows

Article ID: Q64504

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows environment, a multicolumn list box is designed to contain homogeneous data. For example, all the data might be "first names." These first names could logically fall into the same column or be in multiple columns. This feature was added to Windows at version 3.0 to enable a list box to be shorter vertically by splitting the data into two or three columns.

More Information:

To create a multicolumn list box, specify the LBS_MULTICOLUMN style when creating the list box. Then the application calls the SendMessage function to send an LB_SETCOLUMNWIDTH message to the list box to set the column width.

When an application sends an LB_SETCOLUMNWIDTH message to a multicolumn list box, Windows does not update the horizontal scroll bar until the a string is added to or deleted from the list box. An application can work around this situation by performing the following six steps when the column width changes:

1. Send the LB_SETCOLUMNWIDTH message to the list box.

2. Send a WM_SETREDRAW message to the list box to turn off redraw.

3. Add a string to the list box.

4. Delete the string from the list box.

5. Send a WM_SETREDRAW message to the list box to turn on redraw.

6. Call the InvalidateRect function to invalidate the list box.

In response, Windows paints the list box and updates the scroll bar.

Windows automatically manages the list box, including horizontal and vertical scrolling and distributing the entries into columns. The distribution is dependent on the dimensions of the list box. Windows fills Column 1 first, then Column 2, and so on. For example, if an application has a list box containing 13 ordered items and vertical space for 5 items, items 1-5 would be in the first column, items 5-10 in the second, and 11-13 in the last column, and item order would be maintained.

A multicolumn list box cannot have variable column widths.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlListbox

INF: LBS_STANDARD List Box Style Documented Incorrectly

Article ID: Q79675

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A table in Section 8.4.3 (on page 8-12) of the Microsoft Windows Software Development Kit (SDK) "Guide to Programming" manual for version 3.0 states that LBS_BORDER is part of the LBS_STANDARD style. This statement is incorrect; WS_BORDER is part of the LBS_STANDARD style.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Showing the Beginning of an Edit Control after EM_SETSEL
Article ID: Q64758

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In a single-line edit control created with the ES_AUTOHSCROLL style, when the EM_SETSEL message is used to select the entire contents of the control, the text in the control is scrolled to the left and the caret is placed at the end of the text. This occurs when the control contains more text than can be displayed at one time. The order of the starting and ending positions specified in the lParam of the EM_SETSEL message makes no difference.

If your application needs to have the entire contents selected and the beginning of the string in view, create the edit control using the ES_MULTILINE style. The order of the starting and ending positions in the EM_SETSEL message is respected by multiline edit controls.

More Information:

Consider the following example, which sets and then selects the text in a single-line edit control created with the ES_AUTOHSCROLL style:

```
//hEdit and szText defined elsewhere
SetWindowText(hEdit, szText);
SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(0x7FFF, 0));
```

According to the documentation for the EM_SETSEL message, the low-order word of lParam specifies the starting position of the selection and the high-order word specifies the ending position. However, a single-line edit control ignores this ordering and always selects the text from the lower position to the higher position.

If the content of the edit control is longer than the control can display, the text is scrolled to the end of the selection, and the caret is positioned there. In some situations, it is necessary to show the beginning of the text after the selection is made with EM_SETSEL. In Windows 3.00, there is no documented method to accomplish this positioning using a single-line edit control.

A multiline edit control, sized to display only one line and created without the ES_AUTOVSCROLL style, will appear to the user as a single-line control. However, this control will respect the order of the start and end positions in the EM_SETSEL message.

In the sample code above, a multiline edit control will select the text from the specified starting position to the specified ending position, regardless of which position is higher. In this example, the text is scrolled to the beginning and the caret is placed there. The

beginning of the selected text is visible in the control.

Note: A multiline edit control uses up slightly more memory in the USER heap than a single-line edit control.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Changing Text Alignment in an Edit Control Dynamiclly
Article ID: Q66942

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A Microsoft Windows edit control aligns its contents to the left or right margins, or centers its contents, depending on the window style of the control. The control styles ES_LEFT, ES_CENTER, and ES_RIGHT specify left-, center-, and right-alignment, respectively.

Only multiline edit controls can be right-aligned or centered. Single-line edit controls are always left-aligned, regardless of the control style given.

Windows does not support altering the alignment style of a multiline edit control after it has been created. However, there are two methods that you can use to cause a multiline edit control in a dialog box to appear to change alignment. Note that in each of these methods, the dialog box that contains the control must be created with the DS_LOCALEDIT style.

Method 1

Create three controls: one left-aligned, one centered, and one right-aligned. Each has the same dimensions and position in the dialog box, but only one is initially made visible.

When the alignment is to change, call ShowWindow() to hide the visible control and to make one of the other controls visible.

To keep the text identical in all three controls, use the EM_GETHANDLE and EM_SETHANDLE messages to share the same memory among all three controls.

Method 2

Initially create a single control. When the text alignment is to change, retrieve location, size, and style bits for the existing edit control. Create a new control with the same size and in the same location, but change the style bits to reflect the new alignment.

Send the EM_GETHANDLE to each control to retrieve a handle to the memory that stores the contents. Send an EM_SETHANDLE to each control to exchange the memory used by each. Finally, destroy the original control.

There is a sample application named EDALIGN in the Software Library that demonstrates each of these methods. EDALIGN can be found in the Software/Data Library by searching on the keyword EDALIGN, the Q number of this article, or S12796. EDALIGN was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

FIX: Dialog Editor Will Not Change Custom Control Style
Article ID: Q67200

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012015

SYMPTOMS
   When editing a dialog box using the Dialog Editor supplied with the
   Windows Software Development Kit (SDK) version 3.0, and the Dialog
   Editor makes a change to the style of a custom control, the control
   is not repainted to reflect the new style; however, the correct
   style bits are stored in the DLG and RES files.

RESOLUTION
   Microsoft has confirmed this to be a problem in the Windows Dialog
   Editor version 3.0. The Software/Data Library contains the file,
   DLGPATCH, which is a patch program that modifies the Dialog Editor
   .EXE file to correct this problem. The patch should be used only on
   the DIALOG.EXE file dated 6/01/90, file size 96720. Do not use it
   on any other version.

   DLGPATCH can be found in the Software/Data Library by searching on
   the word DLGPATCH, the Q number of this article, or S12797.
   DLGPATCH was archived using the PKware file-compression utility.

   This problem was corrected in the Windows Dialog Editor version 3.1.

Additional reference words: 3.00 softlib MICS3 T5 DLGPATCH.ZIP
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Limiting the Number of Entries in a List Box
Article ID: Q78241

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Although there is no single message that restricts the number of entries (lines) allowed in a list box, the limit can be imposed through the use of subclassing.

More Information:

The following code fragment is an excerpt from a subclassing function that can be used to restrict the number of entries in a list box to no more than the constant MAXENTRIES where the lpfnOldLBFn variable points to the original list box window procedure:

```
long FAR PASCAL SubClassFn(hWnd, message, wParam, lParam)
HWND hWnd;
unsigned message;
WORD wParam;
LONG lParam;
{
   int iCount;

   switch (message)
      {
   case LB_ADDSTRING:
   case LB_INSERTSTRING:
      iCount = SendMessage(hWnd, LB_GETCOUNT, 0, 0L);
      if (iCount > MAXENTRIES)
         { /* Insert action here to inform user of limit violation */
         break;
         }
      /* fall through if less entries than maximum */

   default:
      return CallWindowProc(lpfnOldLBProc, hWnd, message, wParam,
            lParam);
      }
}
```

Additional reference words: 3.00 list box
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Freeing Resources Used by a Multiline Edit Control

Article ID: Q74224

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application developed for the Microsoft Windows graphical environment must explicitly free the resources that a multiline edit control uses in the following two situations:

1. Before sending the edit control a WM_SETHANDLE message to change the text buffer, the application must first send a WM_GETHANDLE message to the control to obtain a handle to the existing buffer. When an application specifies a new text buffer, Windows does not automatically free the existing buffer. To prevent memory loss, the application must free the buffer by calling the LocalFree function.

2. If an application creates a font object and uses the WM_SETFONT message to specify that an edit control should use that font, the application must call the DeleteObject function to delete the font object when it is no longer needed.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlEdit

FIX: Edit Notification Codes Never Received
Article ID: Q65193

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012011

SYMPTOM
When running an application that uses the edit control notification codes declared in the WINDOWS.INC file of the Microsoft Windows Software Developers Kit (SDK) version 3.0, the notifications are never received by the application.

CAUSE
The edit control notifications in WINDOWS.INC are declared incorrectly.

RESOLUTION
Microsoft has confirmed this to be a problem in the Windows SDK version 3.0. To correct the problem, append an "h" to each edit control notification constant. The following is the corrected version of lines 1760-1767 of WINDOWS.INC:

```
; Edit Control Notification Codes
EN_SETFOCUS        = 0100h
EN_KILLFOCUS       = 0200h
EN_CHANGE          = 0300h
EN_UPDATE          = 0400h
EN_ERRSPACE        = 0500h
EN_MAXTEXT         = 0501h
EN_HSCROLL         = 0601h
EN_VSCROLL         = 0602h
```

This problem was corrected in the Windows SDK version 3.1.

Additional reference words: SR# G900823-160 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Determining the Number of Visible Items in a List Box

Article ID: Q78952

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To determine the number of items that are currently visible in a list box, an application must consider the following cases:

1. There are many items in the list box (some items are not visible).

2. There are few items in the list box (the bottom area of the list box is empty).

3. The heights of the items may vary (an owner-draw list box or use of a font other than the system default).

More Information:

The following code segment can be used to determine the number of items visible in a list box:

```
int ntop, nCount, nRectheight, nVisibleItems;
RECT rc;

ntop = SendMessage(hwndList, LB_GETTOPINDEX, 0, 0);
                               // Top item index.

nCount = SendMessage(hwndList, LB_GETCOUNT, 0, 0);
                               // Number of total items.

GetClientRect(hwndList, &rc);
                               // Get list box rectangle.

nRectheight = rc.bottom - rc.top;
                               // Compute list box height.

nVisibleItems = 0;             // Initialize counter.

while ((nRectheight > 0) && (ntop < nCount))
                        // Loop until the bottom of the list box
                        // or the last item has been reached.
   {
   SendMessage(hwndList, LB_GETITEMRECT, ntop, (DWORD)(&itemrect));
                               // Get current line's rectangle.

   nRectheight = nRectheight - (itemrect.bottom - itemrect.top);
                               // Subtract current line height.

   nVisibleItems++;            // Increase item count.
   ntop++;                     // Move to the next line.
```

```
    }
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Size Limits for a Multiline Edit Control

Article ID: Q74225

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows graphical environment, the amount of text that a user can enter into a multiline edit control is limited by the number of characters the user can type into the edit control (set using the EM_LIMITTEXT message) and by the size of the buffer the edit control uses to hold the text.

In general, Windows edit controls were designed as vehicles into which the user can enter and edit small amounts of text. They cannot be used as large-scale text editors.

More Information:

Initially, the user can enter a maximum of 30,000 bytes into a multiline edit control. If the user attempts to enter more text, the edit control beeps and does not accept the characters. An application can set this limit to any value between 1 and 65,535 (0xFFFF) characters by sending the edit control an EM_LIMITTEXT message.

A multiline edit control is also subject to the following limitations:

1. The maximum number of characters in a single line is 1024.

2. The maximum width of a line is 30,000 pixels.

3. The maximum number of lines is approximately 16,350.

By default, the edit control's text buffer is allocated from the application's local heap. Windows can dynamically grow and shrink the text buffer as the user enters text into and deletes text from the edit control. The amount of text that can be edited is determined by how large a buffer Windows can allocate from the local heap. Because the heap shares the application's default data segment with many other objects, the maximum size of the text buffer is likely to be substantially smaller than 64K.

An application can specify a global text buffer for an edit control. By using a global buffer, an edit control can store almost 64K of data. The actual size limit of an edit control depends on the number of lines stored in the edit control. Edit controls contain a dynamically allocated buffer, which contains offsets into the text buffer for each line. Because each line requires 2 bytes of storage, the buffer grows as the number of lines in the edit control grows. In addition to the buffer, there are slightly less than 100 bytes of fixed overhead associated with an edit control. Windows does not

provide any built-in method to process a single block of more than 64K
of text.

One major drawback of using a global memory buffer is that the
EM_GETHANDLE and EM_SETHANDLE messages cannot be used to change the
memory used by the edit control.

For more information on using a global memory buffer, query the
Microsoft Knowledge Base on the following words:

   prod(winsdk) and GLBEDIT

Editing features such as Cut and Paste do not affect the amount of
text that can be edited because Windows uses global memory to
implement these features.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlEdit

FIX: Cannot Change Single-Line Edit Control Color

Article ID: Q67204

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID WIN9012016

SYMPTOMS

When a WM_CTLCOLOR message is sent to a single-line edit control, the color of the interior is not changed properly.

CAUSE

The edit control is painted twice: first as specified in the WM_CTLCOLOR message and again using the default colors.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. There are two methods to work around this problem:

- Create a multiline edit control that is one line in height. Note: Do not include ES_AUTOVSCROLL as part of the edit control style or the edit control will accept multiple lines of input.

-or-

- Add the following code to the WM_CTLCOLOR processing (hBackgroundBrush is the handle to a brush of the color desired for the edit control background):

```
if (CTLCOLOR_MSGBOX == HIWORD(lParam))
   return (DWORD)hBackgroundBrush;
```

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00 SR# G901025-163
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Overcoming the 64 Kilobyte Limit for List Box Data

Article ID: Q79055

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article discusses a technique to overcome the 64K size limit for strings in a list box. It does not discuss many of the implementation details of owner-draw list boxes. For more information about these implementation details, please query on the following words:

   prod(winsdk) and odlist

Standard list box controls are limited to 64K of data. An application can overcome this barrier by implementing an owner-draw list box and managing the data. This can be accomplished by using the list box to store an index to the data, rather than storing the text of each string.

More Information:

In a standard list box control, the data strings are stored in a global memory block that is allocated and managed by Windows. As the number of strings in the list box grows, Windows increases the size of the memory block as necessary to accommodate the data. However, this block is limited to 64K.

One method to overcome this barrier is to implement an owner-draw list box. For each string visible in the list box, Windows sends a WM_DRAWITEM message to the window procedure of the parent window of an owner-draw list box. When an application processes a WM_DRAWITEM message, it displays the string.

For discussion purposes, assume that the application has an index into a database of information. Each index identifies a string to be displayed in the list box. When the application creates the list box, it is important that the LBS_HASSTRINGS style NOT be specified. Although the list box displays strings, the data stored in the list box is a numeric index to the strings, not the strings themselves. Add the data to the list box using a loop with the following statements:

```
   SendMessage(hListbox, LB_ADDSTRING, 0, lMyLongIndex);
```

The value lMyLongIndex is a value of type LONG that identifies the string.

When the parent window's window procedure receives a WM_DRAWITEM message, run the following code:

```
   lMyLongIndex = (LONG)(((LPDRAWITEMSTRUCT)lParam)->itemData);
   GetString(lMyLongIndex, szString);
```

```
   ...

   // Other WM_DRAWITEM processing

   ...

   DrawText(hDC, szString, ...);
```

This code retrieves the index of the selected item from the DRAWITEMSTRUCT structure pointed to by lParam. GetString() is an application-defined function that uses the index to retrieve the string and place it into szString. Finally, the DrawText() function is used to draw the string into the correct display context (identified by hDC).

This method adds one level of indirection to each access of the list box data. By storing an index to each string in the list box, rather than storing the strings themselves, the amount of memory required to maintain the strings in the list box is dramatically reduced. A list box is limited to 64K of data. Storing a long value for each entry allows up to 8160 entries in the list box. Each entry requires 8 bytes (4 bytes for the long value and 4 bytes of overhead) and the list box itself requires 256 bytes for overhead.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: The Parts of a Windows Combo Box and How They Relate

Article ID: Q65881

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A Windows combo box is a compound structure composed of individual windows. Three types of windows can be created as part of a combo box:

1. A combo box itself, of window class "ComboBox"

2. An edit control, of window class "Edit"

3. A list box, of window class "ComboLBox"

The relationship among these three windows varies depending upon the different combo box styles.

More Information:

For combo boxes created with the CBS_SIMPLE styles, the ComboBox window is the parent of the edit control and the list box that is always displayed on the screen. When GetWindowRect() is called for a combo box of this style, the rectangle returned contains the edit control and the list box.

Combo boxes created with the CBS_DROPDOWNLIST style have no edit control. The region of the combo box that displays the current selection is in the ComboBox window itself. When GetWindowRect() is called for a combo box of this style, the rectangle returned does not include the list box.

For combo boxes created with the CBS_DROPDOWN style, three windows are created. The combo box edit control is a child of the ComboBox window. When GetWindowRect() is called for a combo box of this style, the rectangle returned does not include the list box.

However, the ComboLBox (list box) window for combo boxes that have the CBS_DROPDOWN or CBS_DROPDOWNLIST style is not a child of the ComboBox window. Instead, each ComboLBox window is a child of the desktop window. This is required so that, when the drop-down list box is dropped, it can extend outside the application window or dialog box. Otherwise, the list box would be clipped at the window or dialog box border.

Because the ComboLBox window is not a child of the ComboBox window, there is no simple method to get the handle of one window, given the other. For example, given a handle to the ComboBox, the handle to any associated drop-down list box is not readily available. The ComboLBox is a private class registered by USER that is a list box with the

class style CS_SAVEBITS.

Additional reference words: control focus release 3.00 MICS3 R1.7
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Combo Box Case Where GetDlgItemText() Parameter Ignored
Article ID: Q79975

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In a dialog box, when the GetDlgItemText() function is used to copy text from the list box portion of a combo box, the nMaxCount parameter to the GetDlgItemText() function is ignored. Memory will be overwritten if the size of the buffer, which is specified by nMaxCount, is smaller than the length of the currently selected item in the list box.

More Information:

In version 3.0 of the "Microsoft Windows Software Development Kit Reference Volume 1," the documentation for GetDlgItemText() states that this function retrieves the caption or text associated with a control in a dialog box. The GetDlgItemText() function copies the text to the buffer specified by the lpString parameter and returns the number of characters it copies. The string to be copied is truncated if the value specified for nMaxCount is less than the actual string length. GetDlgItemText() sends a WM_GETTEXT message to the child window control.

The documentation for WM_GETTEXT states that in list boxes, the text retrieved is the currently selected item, and wParam specifies the maximum number of bytes to be copied including a null character to terminate the string.

However, the WM_GETTEXT message generated by GetDlgItemText() is translated by the combo box window procedure (which is internal to Windows) to LB_GETTEXT where wParam is set to the index of the item currently selected in the list box.

The documentation for LB_GETTEXT states that the buffer must be large enough to receive the currently selected string and a null character to terminate the string. Therefore, because nMaxCount is ignored, if the buffer provided in GetDlgItemText() is smaller than the currently string, it will overwrite memory, which might cause a variety of difficulties.

If the combo box (or a list box) is the child of the application's main window, instead of the child of a dialog box, then the application can use GetWindowText() to retrieve strings from the combo box. GetWindowText() makes an internal call, which respects the value of nMaxCount and will truncate the string accordingly, instead of placing LB_GETTEXT into the message queue.

Additional reference words: 3.00

KBCategory:
KBSubcategory: UsrCtlCombo

INF: Action of Static Text Controls with Mnemonics

Article ID: Q65883

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The text of a static control may contain a mnemonic, which is a character with which the user can access the control. A mnemonic is indicated to the user by underlining the character in the text of the control, and is created by preceding the desired character with an ampersand (&).

Mnemonic characters are used in conjunction with the ALT key to allow quick access to a control with the keyboard. When the user enters the key combination of the ALT key and the mnemonic character, Windows sets the input focus to the corresponding control and performs the same action as when the mouse is clicked on that control. Push buttons, option buttons, and check boxes all behave in this manner.

Because static text controls do not accept the focus, the behavior of a mnemonic in a static text control is different. When the user enters the mnemonic of a static text control, the focus is set to the next enabled nonstatic control. A static text control with a mnemonic is primarily used to label an edit control or list box. When the user enters the mnemonic, the corresponding control gains the focus.

In this context, the order in which windows are created is important. In a dialog box template, the control defined on the line following the static text control is considered to be "next."

When the user enters the mnemonic of a static text control and the next control is either another static text control or a disabled control, Windows searches for a control that is nonstatic and enabled. In some cases, it may be preferable to disable the mnemonic of a static text control when the control it labels is also disabled. For more information, please query in the Microsoft Knowledge Base on the following word:

mnemonic

More Information:

The dialog box described by the following dialog box template might be displayed by an application when the user chooses Open from the File menu:

```
IDD_FILEOPEN DIALOG LOADONCALL MOVEABLE DISCARDABLE 9, 22, 178, 112
CAPTION "File Open..."
STYLE WS_CAPTION | DS_MODALFRAME | WS_SYSMENU | WS_VISIBLE | WS_POPUP
BEGIN
```

```
   CONTROL "File&name:", ID_NULL, "static",
           SS_LEFT | WS_GROUP | WS_CHILD, 5, 5, 33, 8
   CONTROL "", ID_NAMEEDIT, "edit",
           ES_LEFT | ES_AUTOHSCROLL | WS_BORDER | WS_TABSTOP
           | WS_CHILD | ES_OEMCONVERT, 40, 4, 90, 12
   CONTROL "Directory:", ID_NULL, "static", SS_LEFT | WS_CHILD,
           5, 20, 35, 8
   CONTROL "", ID_PATH, "static", SS_LEFT | WS_CHILD, 40, 20, 91, 8
   CONTROL "&Files:", ID_NULL, "static", SS_LEFT | WS_GROUP
           | WS_CHILD, 5, 33, 21, 8
   CONTROL "", ID_FILELIST, "listbox", LBS_NOTIFY | LBS_SORT
           | LBS_STANDARD | LBS_HASSTRINGS | WS_BORDER | WS_VSCROLL
           | WS_TABSTOP | WS_CHILD, 5, 43, 66, 65
   CONTROL "&Directories:", ID_NULL, "static", SS_LEFT | WS_GROUP
           | WS_CHILD, 75, 33, 49, 8
   CONTROL "", ID_DIRLIST, "listbox", LBS_NOTIFY | LBS_SORT
           | LBS_STANDARD | LBS_HASSTRINGS | WS_BORDER | WS_VSCROLL
           | WS_TABSTOP | WS_CHILD, 75, 43, 65, 65
   CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_TABSTOP
           | WS_CHILD, 139, 4, 35, 14
   CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_TABSTOP
           | WS_CHILD, 139, 23, 35, 14
END
```

In this dialog box, one static text control, with identifier ID_PATH, is used to display the current path. The other four static text controls label other controls, as follows:

```
   "File&name"         labels the ID_NAMEEDIT edit control
   "Directory"         labels the ID_PATH static control display
   "&Files"            labels the ID_FILELIST list box
   "&Directories"      labels the ID_DIRLIST list box
```

When the user enters the key combination ALT+N, Windows sets the focus to the edit control identified in the dialog template as ID_NAMEEDIT, because it is the next enabled nonstatic control. If that edit control was disabled by the EnableWindow function, pressing ALT+N would move the focus to the next enabled nonstatic control. This control would be the list box identified as ID_FILELIST.

Note that the static control "Directory" has no mnemonic; therefore, keyboard input does not affect it.

When the user enters ALT+F, the focus moves to the ID_FILELIST list box, if it is enabled. In the same manner, ALT+D moves the focus to the ID_DIRLIST list box.

If ID_DIRBOX is disabled, ALT+D moves the focus to the OK button, the next enabled nonstatic control. Windows treats this as if the user pressed and released the mouse button over the OK button. For more information on how to prevent this behavior, query the Microsoft Knowledge Base on the following word:

   mnemonic

Additional reference words: 3.00 3.10 radio shortcut
KBCategory:

KBSubcategory: UsrCtlStatic

PRB: Multiline Edit Controls UAE If Resizing Changes LineCount

Article ID: Q66362

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

Multiline edit controls in Windows version 3.0 may cause an unrecoverable application error (UAE) when a key is pressed immediately after the control is resized.

CAUSE

The error will occur only in edit controls with wordwrap turned on. If resizing the control causes the number of lines to decrease, and the caret is positioned in the last line of text, pressing a key will result in a UAE.

RESOLUTION

To correct the error, the edit control must perform some action that updates the caret position. For example, the following line of code corrects the problem and should be used after the control has been resized:

```
SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(32767, 32767));
```

This has the side-effect that the caret is forced to the last character in the control.

More Information:

When the edit control is resized, the number of lines decreases; however, Windows does not update the line position of the caret. If the caret is on the last line, the line number of the caret's position will be greater than the number of lines in the edit control.

When another character is pressed, Windows performs some internal maintenance for line information and uses (number of lines - caret line), which in this case is negative. In the end, this negative number causes Windows to access memory outside of an array boundary.

Using EM_SETSEL forces Windows to update the internal position of the caret. The value 32767, used for the caret position, indicates the "last character" in the control. Because the start and end positions of the EM_SETSEL message are the same, the caret stays in the same position.

Any other caret-updating action on the edit control will also correct the error.

Additional reference words: 3.00

KBCategory:
KBSubcategory: UsrCtlEdit

PRB: Messages Processed by Single & Multiline Edit Controls
Article ID: Q66363

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There are several errors in the "Microsoft Windows Software Development Kit Reference Volume 1" version 3.00, regarding the processing of edit control messages.

The following lists enumerate the messages processed by both single-line and multiline edit controls, and those processed only by multiline controls.

Messages that are documented incorrectly are marked with a pair of asterisks (**), and a comment on the error is given. In short, these messages are EM_GETLINE, EM_GETHANDLE, and EM_LINEFROMCHAR.

More Information:

Messages Processed by Single-Line and Multiline Edit Controls
----

```
    EM_CANUNDO:
    EM_EMPTYUNDOBUFFER:
  **EM_GETLINE:          ;Single-line controls process this message.
    EM_GETMODIFY:
    EM_GETRECT:
    EM_GETSEL:
    EM_LIMITTEXT:
    EM_LINELENGTH:
    EM_REPLACESEL:
    EM_SETMODIFY:
    EM_SETPASSWORDCHAR:
    EM_SETSEL:
    EM_UNDO:
```

Multiline Controls

```
    EM_FMTLINES:
  **EM_GETHANDLE:        ;Single-line controls do not process this message.
    EM_GETLINECOUNT:
  **EM_LINEFROMCHAR:     ;Single-line controls do not process this message.
    EM_LINEINDEX:
    EM_LINESCROLL:
    EM_SETHANDLE:
    EM_SETRECT:
    EM_SETRECTNP:
    EM_SETTABSTOPS:
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

PRB: CBN_SELCHANGE Documented Incorrectly

Article ID: Q80121

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In version 3.0 of the "Microsoft Windows Software Development Kit Reference Volume 1," page 6-17, the documentation for the CBN_SELCHANGE notification code states the following:

> This code indicates that the selection in the list box of a combo box has changed either as a result of the user clicking in the list box or entering text in the edit control. The parent window receives this code through a WM_COMMAND message from the control.

This statement is incorrect; the parent window does not receive a WM_COMMAND message with CBN_SELCHANGE code if the user enters text in the edit control.

More Information:

The parent window receives a WM_COMMAND message with the CBN_SELCHANGE notification code when the user clicks the mouse in the list box or presses the UP ARROW or DOWN ARROW key to change the selection in the list box.

When the user enters text in the edit control, the parent window receives the CBN_EDITCHANGE and CBN_EDITUPDATE notification codes through WM_COMMAND messages from the control.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Processing CBN_SELCHANGE Notification Message

Article ID: Q66365

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When a combo box receives a CBN_SELCHANGE notification message, GetDlgItemText() will give the text of the previous selection and not the text of the new selection.

To get the text of the new selection, send the CB_GETCURSEL message to retrieve the index of the new selection and then send a CB_GETLBTEXT message to obtain the text of that item.

More Information:

When an application receives the CBN_SELCHANGE notification message, the edit/static portion of the combo box has not been updated. To obtain the new selection, send a CB_GETLBTEXT message to the combo box control. This message places the text of the new selection in a specified buffer. The following is a brief code fragment:

```
   ...  /* other code */

   case CBN_SELCHANGE:
      hCombo = LOWORD(lParam);  /* Get combo box window handle */

    /* Get index of current selection and then the text of that selection */

      index = SendMessage(hCombo, CB_GETCURSEL, (WORD)0, 0L);
      SendMessage(hCombo, CB_GETLBTEXT, (WORD)index, (LONG)buffer);
      break;

   ...  /* other code */
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Sample Code Demonstrates Changing Size of an Edit Control
Article ID: Q80553

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

EXPEDIT is a file in the Software/Data Library that demonstrates changing the size of an edit control in response to an EN_UPDATE message. In the sample, when the typed text reaches the end of a line, the edit control expands to allow the user to continue typing on a new line.

EXPEDIT can be found in the Software/Data Library by searching on the word EXPEDIT, the Q number of this article, or S13280. EXPEDIT was archived using the PKware file-compression utility.

More Information:

Windows sends the EN_UPDATE notification message to the parent of the edit control whenever the contents of the edit control have been altered, but before the display is updated. The notification is sent to the parent window as the HIWORD of the lParam parameter of a WM_COMMAND message. The ID of the control is in the wParam.

Therefore, for an application to determine when it is necessary to change the size of the edit control, it must handle the WM_COMMAND case in which the wParam is the ID of the control and the HIWORD of the lParam is EN_UPDATE. When EXPEDIT receives an EN_UPDATE notification message, it performs the following eight actions:

1. Sends an EM_FMTLINES message to the edit control (wParam = 1). When the control processes this message, it modifies its buffer to have the character sequence carriage return-carriage return-linefeed (CR-CR-LF) at the end of each line.

2. Retrieves the length of the edit control's buffer.

3. Passes a handle to the (nonempty) buffer of the DrawText() function, which calculates the size of the rectangle required to hold all the text.

4. Adjusts the text height returned by DrawText because DrawText interprets the CR-CR-LF sequence at the end of each buffer line as two lines instead of one.

5. Compares the right edge of the rectangle returned from DrawText with the "limit." DrawText will extend the right edge of the rectangle if the text is too long. If the right edge of the last line exceeds the limit, then the edit control needs an additional line. EXPEDIT performs all the obvious limit checks.

6. If the height has not changed, then it is not necessary to change the size of the edit control, and EXPEDIT returns. Otherwise, the size of the edit control must be changed.

7. Updates the size of the edit control with MoveWindow() and UpdateWindow().

8. Returns FALSE so that the Dialog Manager will not do anything when it receives the EN_UPDATE message.

Note: The key idea of this implementation is to "predict" whether the next character will fit in the edit control; if it will not, then add a new line to the control so that the character will fit. This prediction is accomplished by subtracting the width of "W" from the edge of the edit control's rectangle to get the limit.

Additional reference words: 3.00 3.0 EXPEDIT.ZIP
KBCategory:
KBSubcategory: UsrCtlEdit

INF: EM_SETTABSTOPS Does Not Free All of Local Heap Memory

Article ID: Q66452

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When an EM_SETTABSTOPS message is sent to an edit control, a 12-byte fixed object remains in the local heap. Even if the window is destroyed, the 12-byte fixed object is still left in the local heap.

By setting the tab stops to 0 (zero) prior to the destroy, the 12-byte fixed object is released.

Below is a sample of how to correct this problem. The first SendMessage() call should be inserted in the WM_CREATE code section, and the second call to SendMessage() and the DestroyWindow() call should be in the WM_CLOSE section.

```
  // During WM_CREATE:
  // This creates a twelve-byte fixed object in the local heap that
  // is not freed!

     SendMessage( hWndChild, EM_SETTABSTOPS, 1, (LONG)(LPINT)&n );

  // During WM_CLOSE:
  // This frees the twelve-byte fixed object that windows leaves
  // in the local heap.

     SendMessage( hWndChild, EM_SETTABSTOPS, 0, (LONG)NULL );
     DestroyWindow(hWndChild);
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Preventing Screen Flash During List Box Multiple Update

Article ID: Q66479

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The WM_SETREDRAW message can be used to set and clear the redraw flag for a window. Before an application adds many items to a list box, this message can be used to turn the redraw flag off, which prevents the list box from being painted after each addition. Properly using the WM_SETREDRAW message keeps the list box from flashing after each addition.

More Information:

The following four steps outline ways to use the WM_SETREDRAW message to facilitate making a number of changes to the contents of a list box in a visually pleasing manner:

1. Clear the redraw flag by sending the list box a WM_SETREDRAW message with wParam set to FALSE. This prevents the list box from being painted after each change.

2. Send appropriate messages to make any desired changes to the contents of the list box.

3. Set the redraw flag by sending the list box a WM_SETREDRAW message with wParam set to TRUE. The list box does not update its display in response to this message.

4. Call InvalidateRect(), which instructs the list box to update its display. Set the third parameter to TRUE to erase the background in the list box. If this is not done, if a short list box item is drawn over a long item, part of the long item will remain visible.

The following code fragment illustrates the process described above:

```
   /* Step 1: Clear the redraw flag. */
   SendMessage(hWndList, WM_SETREDRAW, FALSE, 0L);

   /* Step 2: Add the strings. */
   for (i = 0; i < n; i++)
      SendMessage(hWndList, LB_ADDSTRING, ...);

   /* Step 3: Set the redraw flag. */
   SendMessage(hWndList, WM_SETREDRAW, TRUE, 0L);

   /* Step 4: Invalidate the list box window to force repaint. */
   InvalidateRect(hWndList, NULL, TRUE);
```

Additional reference words: 3.00 SR# G900921-44 flash flicker
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Changing the Color of an Edit Control

Article ID: Q74043

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The source code fragment below demonstrates how to paint the foreground and background of a single-line edit control in an application developed for the Microsoft Windows graphical environment. It creates a window that contains one single-line edit control and paints the edit control cyan on blue.

Windows sends the WM_CTLCOLOR message to the parent of an edit control before the control is painted. If desired, the parent window, usually a dialog box procedure, can process this message and change the text and background colors of the control.

Note: Windows version 3.0 does not properly change the background color of a single-line edit control. For more information on that problem, query in the Microsoft Knowledge Base on the following words:

   prod(winsdk) and WM_CTLCOLOR and single and edit

The following MainWndProc demonstrates WM_CTLCOLOR processing:

```
long FAR PASCAL MainWndProc(HWND hwnd, unsigned msg,
                            WORD wParam, LONG lParam)
{
static   HWND     hwndEdit;
static   HBRUSH   hBrush;

   switch (msg)
   {
      case WM_CREATE:
         {
            HDC hdc;
            TEXTMETRIC tm;

            hdc = GetDC(hwnd);
            GetTextMetrics(hdc, &tm);
            ReleaseDC(hwnd, hdc);

            hwndEdit = CreateWindow("edit", NULL,
                            WS_CHILD | WS_VISIBLE
                            | ES_AUTOHSCROLL | ES_MULTILINE,
                            10, 20, 100, tm.tmHeight,
                            hwnd, 1, hInst, NULL);

            // Create a blue brush to be used for the edit control's
            // background color.
```

```
         hBrush = CreateSolidBrush(RGB(0, 0, 255));
      }
      break;

    case WM_CTLCOLOR:
    {
      // Set foreground and background colors only if this
      // is an edit control.
      if (HIWORD(lParam) == CTLCOLOR_EDIT)
      {
         // Set the edit control's foreground text color to
         // cyan and the text's background color to blue.
         SetTextColor(wParam, RGB(0, 255, 255));
         SetBkColor(wParam, RGB(0, 0, 255));

         // Properly originate the background brush. This is
         // of use if the brush is a pattern instead of a
         // solid color.
         UnrealizeObject(hBrush);
         SetBrushOrg(wParam, 0, 0);

         // Return a handle to the background brush for the edit
         // control.
         return (DWORD)hBrush;
      }
    }
    break;

    case WM_SETFOCUS:
      SetFocus(hwndEdit);
      return 0;

    case WM_DESTROY:
      PostQuitMessage(0);
      DeleteObject(hBrush);
      break;
  }
  return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Setting Tab Stops in a Windows 3.0 List Box
Article ID: Q66652

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Tab stops can be used in a list box to align columns of information. This article describes how to set tab stops in a list box and provides a code example that demonstrates the process.

More Information:

To set tab stops in a list box, perform the following three steps:

1. Specify the LBS_USETABSTOPS style when creating the list box.

2. Assign the desired tab stops to an integer array.

   a. The tab stop values must be in increasing order -- back tab stops are not allowed. The tabs work the same as typewriter tabs: once a tab stop is overrun, a tab character will move the cursor to the next tab stop. If the tab stop list is overrun (that is, the current position is greater than the last tab stop value), the default tab of eight characters is used.

   b. The tab stops should be specified in dialog units. On the average, each character is about four horizontal dialog units in width.

   c. It is possible to hide columns of text from the user by specifying tab stops beyond the right side of the list box. This can be a useful way to hide information used for the application's internal processing.

3. Send an LB_SETTABSTOPS message to the list box to set the tab stops. For example:

   ```
   SendMessage(GetDlgItem(hDlg, IDD_LISTBOX),
               LB_SETTABSTOPS,
               TOTAL_TABS,
               (LONG)(LPSTR)TabStopList);
   ```

   a. If wParam is set to 0 (zero) and lParam to NULL, the tab stops are set to two dialog units by default.

   b. SendMessage() will return TRUE if all of the tab stops are set successfully; otherwise, SendMessage() returns FALSE.

Example

Below is an example of the process. Tab stops are set at character positions 16, 32, 58, and 84.

```
   int     TabStopList[TOTAL_TABS]; /*  Array to store tabs */

   TabStopList[0] = 16 * 4;          /* 16 spaces */
   TabStopList[1] = 32 * 4;          /* 32 spaces */
   TabStopList[2] = 58 * 4;          /* 58 spaces */
   TabStopList[3] = 84 * 4;          /* 84 spaces */

   SendMessage(GetDlgItem(hDlg, IDD_LISTBOX),
               LB_SETTABSTOPS,
               TOTAL_TABS,
               (LONG)(LPSTR)TabStopList);
```

If the desired unit of measure is character position, then specifying tab positions in dialog units is recommended. Dialog units are independent of the current font; they are loosely based on the average width of the system font. Each character takes approximately four dialog units.

For more control over the exact placement of a tab stop, the desired position should be converted to a pixel offset and this offset should be converted into dialog units. The following formula will take a pixel position and convert it into the first tab stop position before (or at) the desired pixel position:

```
   TabStopList[n] = 4 * DesiredPixelPosition /
                    LOWORD(GetDialogBaseUnits());
```

There is a sample application named TABSTOPS in the Software/Data Library that demonstrates how tab stops are set and used in a list box. TABSTOPS can be found in the Software/Data Library by searching on the word TABSTOPS, The Q number of this article, or S12785. TABSTOPS was compressed using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Multiline Edit Control Does Not Show First Line

Article ID: Q66668

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When a multiline edit control is created that is less than one system character in height, the text in the edit control will not be displayed and subsequent attempts to enter text will cause the edit control to beep. This functionality is an invalid multiline edit control under Microsoft Windows version 3.00, even though this construct does work in Windows versions 2.x.

The multiline edit control also checks to see if the next line of text is displayable. If the next line of text is not displayable, it will beep to let you know that you have reached the limit of the edit control.

There is a similar situation with a control that overlaps another control in a dialog box. This construct is also considered invalid; thus, the second control will not be displayed.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Implementing a Read-Only Edit Control In Windows
Article ID: Q80946

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In some situations, an application may contain text that is displayed for the user to read, which is undesirable for the user to modify. The application can use a static control to contain this text if the message is short. However, for larger amounts of text, which require scrolling to display all the text in the allotted area, something closer to an edit control is required. This article, and its associated sample code, details how to make an edit control "read-only."

ROEDIT is a file in the Software/Data Library that demonstrates the techniques discussed in this article. ROEDIT can be found in the Software/Data Library by searching on the word ROEDIT, the Q number of this article, or S13284. ROEDIT was archived using the PKware file-compression utility.

More Information:

An application can create a read-only edit control by subclassing or superclassing a standard edit control. The subclass procedure filters out messages that change the contents of the edit control. The following code fragment demonstrates this process:

```
   FARPROC gOldProc;

   LONG FAR PASCAL ROEditProc(HWND hWnd, WORD msg,
                              WORD wParam, LONG lParam)
   {
       switch (msg)
       {
           case WM_KEYUP:
           case WM_KEYDOWN:
           case WM_CHAR:
           case WM_CUT:
           case WM_COPY:
           case WM_PASTE:
           case WM_LBUTTONUP:
           case WM_LBUTTONDOWN:
           case WM_LBUTTONDBLCLK:
               return 1L;

           case WM_GETDLGCODE:
               return 0L;
       }
```

```
        return CallWindowProc(gOldProc, hWnd, msg, wParam, lParam);
    } //*** ROEditProc
```

In the example above, the subclass procedure traps mouse clicks, keystrokes, and the cut, copy, and paste commands. It also traps the WM_GETDLGCODE message to prevent an edit control in a dialog box from receiving the input focus.

The following example demonstrates superclassing an edit control to create a new ROEDIT-class control that behaves similar to a read-only edit control. A ROEDIT control implements the window procedure provided above. It also changes the cursor to an arrow instead of an I-beam, which provides an additional indication to the user that the contents of the control cannot be changed. Using an ROEDIT control eliminates the necessity of subclassing each control after it is created. When the application creates a control from the ROEDIT class, the read-only behavior is provided automatically.

The following code demonstrates superclassing an edit control as described above:

```
    FARPROC gOldProc;

    BOOL RegisterROEdit(HANDLE hInstance)
    {
        WNDCLASS wc;

        if (!GetClassInfo(NULL, "EDIT", &wc))
            return FALSE;

        gOldProc = (FARPROC)wc.lpfnWndProc;

        wc.style        &= ~CS_GLOBALCLASS;
        wc.lpfnWndProc   = ROEditProc;
        wc.hInstance     = hInstance;
        wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wc.lpszClassName = "ROEDIT";

        return RegisterClass(&wc);

    } //*** RegisterROEdit
```

In Windows 3.1, there is a new style bit for the edit control (ES_READONLY) that removes the editing capabilities of the edit control, leaving only the viewing capabilities. This style is useful when the application shows the user a body of static text that the user reads and does not modify.

Additional reference words: 3.00 3.10 3.x softlib ROEDIT.ZIP
KBCategory:
KBSubcategory: UsrCtlEdit

PRB: Custom Control Documentation Errors
Article ID: Q67243

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following is a list of documentation errors in chapter 20 of the "Microsoft Windows Software Development Kit Guide to Programming" version 3.0:

1. Page 20-11: Table of exported functions and their ordinal values is misleading.

   It is not necessary to export the ClassInit() [or LibMain()] function. This function is called only once when the DLL is loaded and is not called by the Dialog Editor or by any other application.

   The ClassWndFn() function is the window procedure of the control class. While this function must be exported, it is not necessary to associate the ordinal value 5 with it. The ClassWndFn() may be exported as any ordinal except 2, 3, 4, or 6.

2. Page 20-13: CTLTYPE pointer types not declared.

   On this page, and in the Windows SDK include file CUSTCNTL.H, there are no definitions of the various types of pointers to the CTLTYPE structure. The following additional type declarations should be included:

   ```
   typedef CTLTYPE *       PCTLTYPE;
   typedef CTLTYPE FAR *   LPCTLTYPE;
   ```

3. Page 20-15: hCtlStyle parameter to ClassStyle() function is a handle to global memory.

   The hCtlStyle parameter to the ClassStyle() function should be documented as a HANDLE to global memory. The ClassStyle() function must call GlobalLock() on this handle to access the data, and must call GlobalUnlock() when the data is no longer needed.

4. Page 20-16: Documentation for the dwStyle field in the CTLSTYLE structure is incorrect.

   The documentation for the dwStyle field incorrectly states that "The high-order word contains the control specific flags, while the low-order word contains the Windows-specific flags."

   The statement in the documentation is backwards. The Windows-specific flags are contained in the high-order word, and the control-specific flags are contained in the low-order word.

5. Page 20-18: Documentation for the dwFlags parameter to the ClassFlags() function is incorrect.

   The first parameter of the ClassFlags() function is documented as a DWORD, dwFlags. The first parameter is actually a WORD parameter, wFlags. wFlags contains only the control-specific flags. The Rainbow sample code provided in the SDK correctly uses a WORD value.

   The description of the ClassFlags function incorrectly states that "This function should not interpret the flags contained in the high-order word since these are managed by the Dialog Editor." There is no need for this statement because there is no high-order word.

6. Page 20-18: Documentation for the wMaxString parameter to the ClassFlags() function is incorrect.

   The documentation for this parameter states that this parameter "Specifies the maximum length of the style ring." The last word should be "string."

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCustomctl

FIX: No Focus Rect on List Box at Dialog Creation
Article ID: Q66995

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows
  version 3.0

SYMPTOMS
========

When a modal dialog box is created, if the initial focus is set to a list box, no focus rectangle is drawn.

RESOLUTION
==========

The steps below detail a method to avoid this problem. When the dialog box is created, the dialog box posts a user-defined message to itself. Processing this message sets the focus to the list box. The following code fragment from a dialog function demonstrates this procedure:

```
   case WM_INITDIALOG:
      SendDlgItemMessage(hDlg, ID_LISTBOX, LB_ADDSTRING, 0,
           (LONG)(LPSTR)"Item 1");
      SendDlgItemMessage(hDlg, ID_LISTBOX, LB_ADDSTRING, 0,
           (LONG)(LPSTR)"Item 2");
      PostMessage(hDlg, WM_USER+1000, 0, 0L);
      return (FALSE);

   case WM_USER+1000:
      SetFocus(GetDlgItem(hDlg, ID_LISTBOX));
      break;
```

NOTE: The dialog box must have been created with the WS_VISIBLE style in order for the workaround suggested above to work properly.

STATUS
======

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00 listbox 3.10
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Controlling the Horizontal Scroll Bar on a List Box
Article ID: Q66370

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

This article discusses everything that is required to fully control the horizontal scroll bar on a list box. This information is also available in TIPS.HLP which included with Microsoft Visual C/C++. The following is an outline of the information presented in this article:

A. Windows support for a horizontal scroll bar on a list box

   1. Overview
   2. Software Library sample code

B. New list box messages: LB_SETHORIZONTALEXTENT and LB_GETHORIZONTALEXTENT

   1. Default setting for extents is zero.
   2. Messages only change internal value -- do not affect the visibility of the scroll bar.

C. Making the scroll bar visible when adding or inserting a string

   1. Must change extent before adding a string.
   2. Must redraw the list box after adding a string.

D. Hiding the scroll bar when deleting a string

   1. Must change the extent before deleting a string.
   2. Must redraw the list box after deleting a string.
   3. Send WM_HSCROLL message with SB_TOP parameter to scroll the list box to the left in case it is scrolled right.
   4. Special Handling is needed for LB_RESETCONTENT.

E. Calculating horizontal pixel extents of strings

   1. Use GetTextExtent function.
   2. Font considerations: use WM_GETFONT message.
   3. Use GetTextMetrics and add one tmAveCharWidth.

F. Maintenance of all extents in list box

G. Sample code

   1. Support functions as an additional C file -- static linking to application.
   2. Support functions in a DLL, multiple list boxes supported.

More Information:

Windows Support for Horizontal Scroll Bar in a List Box

Microsoft Windows version 3.0 is the first version that recognizes the WS_HSCROLL window style for list boxes. This style adds a horizontal scroll bar to the list box. However, the scroll bar does not appear automatically when a string that is too long to display in a list box is added to that list box. Similarly, when the last string longer than the list box is removed, Windows will not hide the scroll bar. This behavior is different from that of the vertical scroll bar of a list box, which is added and removed as needed.

An application must maintain the width, in pixels, of each string in the list box. The LB_SETHORIZONTALEXTENT list box message controls both the scrolling range and the visibility of a horizontal scroll bar. In this article, the term "extent" is defined to be the width of an object in screen pixels. Each string has an extent as does the list box itself.

The other sections of this article provide more detailed information about special considerations that must occur when dealing with horizontal scroll bars in list boxes.

There are two files in the Software Library that demonstrate the points made in this article. Each file contains a set of horizontal scroll bar support functions that maintain the extents of all strings in the list box and change the scrollable list box extent as required. Additional details on these samples are found in the "Sample Code" section below.

The LISTHORZ file contains a complete sample application demonstrating a list box with a horizontal scroll bar. The necessary support functions are contained in a C module that can be compiled and linked to any Windows application. LISTHORZ can be found in the Software/Data Library by searching on the word LISTHORZ, the Q number of this article, or S12804.

The LISTHSCR file contains the complete sources for a DLL that contains the necessary list box support functions. These functions are exactly the same as those in LISTHORZ. Included in this archive is an application that uses the services of the DLL to perform the same functions as the application in LISTHORZ. LISTHSCR can be found in the Software/Data Library by searching on the word LISTHSCR, the Q number of this article, or S13558.

LISTHORZ and LISTHSCR were archived using the PKware file-compression utility.

New List Box Messages:
LB_SETHORIZONTALEXTENT and LB_GETHORIZONTALEXTENT

Two messages have been added to Windows 3.0 to support horizontal scroll bars in list boxes:

```
   Message                   Description
   -------                   -----------

   LB_SETHORIZONTALEXTENT  Sets the width in pixels by which a list
                           box can be scrolled to the value of wParam
                           in the message

   LB_GETHORIZONTALEXTENT  Retrieves the width in pixels by which a
                           list box can be scrolled
```

By default, the horizontal scroll bar extent of a list box is 0 (zero). Because 0 is less than the width of the list box client area, Windows will not add a scroll bar to the window until the scroll bar extent is changed to a value larger than the list box extent.

However, the LB_SETHORIZONTALEXTENT message does not affect the visibility of a horizontal scroll bar. If a scroll bar is visible, sending this message with a small extent specified will not remove it. Likewise, if a scroll bar is not present, sending this message with a large extent will not create one.

The next two sections of this article explain how to add and remove a horizontal scroll bar as strings are added and deleted. The major point is that Windows will only show or hide the scroll bar when a string is added, inserted, or deleted.

Making the Scroll Bar Visible When Adding or Inserting a String

When a string with an extent larger than the width of the list box is to be added, an application must send the LB_SETHORIZONTALEXTENT message before sending an LB_ADDSTRING or LB_INSERTSTRING message.

During the process of adding or inserting a string, Windows compares the horizontal scroll bar extent stored in the list box to the width of the list box client area. If the client area extent is smaller than the scroll bar extent, the scroll bar is made visible the next time the list box is drawn.

The client area width of the list box does not include the width of the vertical scroll bar, if it is visible. Consider a list box without a vertical scroll bar, that is filled with strings. Each of these strings is slightly narrower than the list box. When another string is added, and the vertical scroll bar is caused to appear, Windows discovers that the horizontal scroll bar extent is now greater than the width of the list box, and adds a horizontal scroll bar.

If the scroll bar extent is less than the width of the client area of the list box, the status of the scroll bar remains unchanged.

If the list box is not drawn after the string is added, the scroll bar will not appear. Therefore, if the WM_SETREDRAW message is used to turn redraw off, adding a string will not show the horizontal scroll bar until the list box is redrawn.

Hiding the Scroll Bar When Deleting a String

Windows only removes the horizontal scroll bar during the processing of an LB_DELETESTRING message. Therefore, if the string to be deleted is the longest in the list box, the horizontal scroll bar extent must be changed to the next smaller extent value before that string is deleted. After the string is deleted, Windows compares the stored scroll bar extent to the width of the client area of the list box, and if the extent is smaller, the scroll bar is removed.

If the list box is not drawn after the string is deleted, the scroll bar will not disappear. Therefore, if the WM_SETREDRAW message is used to turn redraw off, deleting a string will not remove the horizontal scroll bar until the list box is redrawn.

However, if the list box is scrolled to the right by as little as one pixel, the scroll bar will remain visible, regardless of the extent that is set. This is done so that the user can always scroll back to the extreme left. If the scroll bar was removed, the list box might be left in a state where it is scrolled to the right by some amount without any way to scroll back completely to the left.

To work around this problem, always scroll the list box to the extreme left before deleting the longest string. If a shorter string is deleted, the extent will remain the same and the horizontal scroll bar will not be affected anyway. Only scroll the list box if the longest string is being deleted.

The list box can be scrolled either completely to the left (in the case where a long string still exists in the list box) or just enough so that the next longest string is visible, assuming that the list box requires a scroll bar. The sample code in the Software Library (described above) always scrolls the list box completely left, using the WM_HSCROLL message, as follows:

```
   SendMessage(hList, WM_HSCROLL, SB_TOP, 0L);
```

The LB_RESETCONTENT message does not affect the state of the horizontal scroll bar even though all strings in the list box are deleted. Before an LB_RESETCONTENT message is sent, an application must perform the following steps:

1. Send a LB_SETHORIZONTALEXTENT message with an extent of 0 (zero).

2. Send a WM_HSCROLL message to scroll the list box completely to the left. This method is given above.

3. Send an LB_DELETESTRING message with an index of 0 (zero). This will delete the first string, if there is one, and will also remove the scroll bar.

If any strings are present in the list box, the first is removed by the LB_DELETESTRING message and the scroll bar is removed, because the extent has been set to zero. The LB_RESETCONTENT message will then remove the remaining strings. If there are no strings in the list box,

the LB_DELETESTRING message will return an error. However, because there are no strings in the list box, there should be no scroll bar in the first place.

Calculating Horizontal Extents of Strings

The previous discussion mentions the extents of strings, but provides no methods for determining these values.

Pixel extents of strings are calculated by the GetTextExtent function. This is a GDI call that sums the pixel width of each character in a string using the font that is currently selected into a display context represented by an hDC.

If lpString holds a representative string and hWndListBox is the window handle to the list box, the following steps are required to determine the size of each string:

1. Declare the following variables:

```
DWORD       dwExtent;
HDC         hDCListBox;
HFONT       hFontOld, hFontNew;
TEXTMETRICS tm;
```

2. Use GetDC to retrieve the handle to the display context for the list box and store it in hDCListBox:

```
hDCListBox = GetDC(hWndListBox);
```

3. Send the list box a WM_GETFONT message to retrieve the handle to the font that the list box is using, and store this handle in hFontNew:

```
hFontNew = SendMessage(hWndListBox, WM_GETFONT, NULL, NULL);
```

4. Use SelectObject to select the font into the display context. Retain the return value from the SelectObject call in hFontOld:

```
hFontOld = SelectObject(hDCListBox, hFontNew);
```

5. Call GetTextMetrics to get additional information about the font being used:

```
GetTextMetrics(hDC, (LPTEXTMETRIC)&tm);
```

6. For each string, the value of the extent to be used is calculated as follows:

```
dwExtent = GetTextExtent(hDCListBox, lpString, strlen(lpString))
       + tm.tmAveCharWidth;
```

7. After all the extents have been calculated, select the old font back into hDCListBox and then release it.

```
      SelectObject(hDCListBox, hFontOld);
      ReleaseDC(hWndListBox, hDCListBox);
```

There is a reason for step 5 and adding the additional width to the string in step 6 above. If the largest value returned by GetTextExtent is used as the parameter in the LB_SETHORIZONTALEXTENT message, the longest string will not show completely when the list box is scrolled completely to the right. A few pixels are clipped by the right border on the list box.

The tmAveCharWidth field in the TEXTMETRIC structure provides a consistent number of pixels to add to the length of the string, no matter what font is presently in use. This buffer space keeps the strings from being clipped.

As a side note, the value of tmAveCharWidth is the number of pixels by which the list box is scrolled horizontally when the scroll bar arrows are clicked. If a fixed pitch font is used, the list box is scrolled horizontally by one character for each click.

Maintenance of All Extents in List Box

Many possible methods can be used to maintain a list of the extents of the strings in the list box. One of the most convenient methods is to use property lists, as shown in the sample code.

Every window has a property list associated with it. Each property is a string and an associated data handle. A window stores and retrieves each data handle using the string that labels the handle.

A sorted list of the extents for strings in the list box can be saved in a local or global memory object. This allows each window to keep its own list and does not require that the application maintain a mapping from the list box to the data handle itself. The list of extents should be sorted in descending order, so that the first extent in the list is that of the longest string in the list box. Keeping this list sorted also allows the application to use a binary search to find an extent in the list when one is being inserted or deleted.

When a new string is added, insert that string's extent into the list, maintaining the sorted order. If the new extent is placed at the top of the list, send an LB_SETHORIZONTALEXTENT message to the list box specifying the new extent. Do not send the message for an extent that is not the largest.

When a string is deleted, remove that string's extent from the list. If the extent is the first in the list, then that string is the longest in the list box. In this case, send an LB_SETHORIZONTALEXTENT to the list box specifying the next largest extent in the list. If a smaller string is deleted, do not send the message.

When an LB_RESETCONTENT message is used, clear the entire list of extents. Send an LB_SETHORIZONTALEXTENT message specifying an extent of 0 (zero), followed by an LB_DELETESTRING as outlined in the last

part of the section above titled "Hiding the Scroll Bar When Deleting a String."

Sample Code

As mentioned above, there are two archives in the Software Library: LISTHORZ and LISTHSCR. Each of these samples provides five support functions that greatly facilitate the maintenance of horizontal scroll bars in list boxes.

In LISTHORZ, these functions are found in the file LISTHELP.C. This file can be compiled separately and linked into an application.

LISTHSCR contains the source files for a DLL that provides these five support functions. The C file for the DLL is exactly the same as in LISTHORZ, except that its name has changed. This archive also includes a LISTHORZ program that uses the services of the DLL to perform the same functions as the program in the other archive. The file LISTHAPI.H is the include file containing prototypes of the functions exported by the DLL. Also included is an import library, LISTHSCR.LIB, that is generated by the makefile.

The DLL is most useful because it can support horizontal scroll bars in any number of list boxes in any number of applications.

The remainder of this article documents the support functions.

BOOL FInitListboxExtents(HWND hList)

Allocates local memory to store a list of string extents for the list box identified by hList. The handle to this local memory is saved in the property list of the list box. This function should be called after the list box is created, such as during WM_INITDIALOG processing.

Parameters:

```
    hList    HWND   Handle to the list box that will use a horizontal
                    scroll bar.
```

Return Value:

```
    BOOL            TRUE if there are no errors, FALSE if memory could
                    not be allocated.
```

BOOL FFreeListboxExtents(HWND hList)

Frees the memory allocated for the extent list of the list box identified by hList. The property that stores the memory handle set by FInitListboxExtents is removed. This function should be called when the list box is being destroyed.

Parameters:

```
    hList    HWND   Handle to the list box that was previously used
                    with FInitListboxExtents

  Return Value:
    BOOL            TRUE if there are no errors, FALSE if memory could
                    not be freed, in which case the property is not
                    removed.
```

----

void ResetListboxExtents(HWND hList)

Removes all previously saved extents in the extent list. This is accomplished by calling FFreeListboxExtents and FInitListboxExtents in succession. The horizontal extent of the list box is set to 0 (zero) and any horizontal scroll bar is removed. This function should be called before an LB_RESETCONTENT message is sent to the list box.

```
  Parameters:
    hList    HWND   Handle to the list box that will be reset.

  Return Value:
    none
```

WORD WAddExtentEntry(HWND hList, LPSTR psz)

Adds an extent entry into the list box's extent list. The extent added is that of the string pointed to by psz using the current font in the list box. If the extent added is larger than any other in the extent list, an LB_SETHORIZONTALEXTENT message is sent to the list box with this new extent.

This function must be called before the string is added to the list box with LB_ADDSTRING or LB_INSERTSTRING.

```
  Parameters:
    hList    HWND   Handle to the list box to which the string is to
                    be added.

    psz      LPSTR  Pointer to string that is to be added. This
                    function must be called before the string is added
                    so that the horizontal scroll bar will be properly
                    maintained.

  Return Value:
    WORD            One of these three values:
                    0 if the string added was not the longest string
                       in the list box and the visibility of the
                       horizontal scroll bar did not change.
                    The extent of the string added, if the added
                       string was the longest and the visibility of
                       the scroll bar may have changed.
                    -1 for an error.
```

```
WORD WRemoveExtentEntry(HWND hList, WORD iSel)

  Removes the extent entry of the string identified by index iSel in
  the list box. If the string to be removed is the longest in the list
  box, the list box is scrolled completely to the left and the
  horizontal extent is set to that of the next longest string.

  This function must be called before an LB_DELETESTRING message is
  sent to the list box.

  Parameters:
    hList    HWND   Handle to the list box from which a string is to
                    be removed.

    iSel     WORD   Index of the string to be removed.

  Return Value:
    WORD            One of these three values:
                    0 if the string removed was not the longest in the
                       list box and the visibility of the horizontal
                       scroll bar did not change
                    The extent of the string deleted, if the deleted
                       string was the longest and the visibility of
                       the scroll bar may have changed.
                    -1 for an error.
```

Additional reference words: 3.00 softlib LISTHORZ.ZIP LISTHSCR.ZIP
LB_SETH LB_GETH

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: MicroScroll Custom Control Code in Software Library
Article ID: Q67247

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Software Library contains a custom control DLL and documentation for a custom control class, MicroScroll. These controls are essentially small scroll bars with no scroll box, and perform no scroll-box tracking.

MicroScroll controls are small enough to be placed next to edit controls to create a spin button, such as those in the Date/Time dialog box in the Control Panel. Spin buttons are discussed on pages 86 and 87 of the IBM CUA guide, which is included with the Windows Software Development Kit (SDK).

The archive file in the Software Library, MUSCROLL, contains the following files:

Filename	Description
MUSCROLL.DLL	MicroScroll custom control DLL. Currently, the source files necessary to build the custom control are not available to the public.
MUSCROLL.H	Header file containing control styles and messages that define the interface to the control.
MUSCROLL.HLP	Windows Help file describing the control interface and notes on using the control in an application.
MUSTEST.ZIP	Code for a test program that uses MicroScroll controls to implement a numeric spin button and to add horizontal scrolling capabilities to a single line edit control.

MUSCROLL can be found in the Software/Data Library by searching on the word MUSCROLL, the Q number of this article, or S12831. MUSCROLL was archived using the PKware file-compression utility.

To use MicroScroll effectively with the Dialog Editor that is shipped with the Microsoft Windows SDK version 3.0, you must first patch the Dialog Editor executable file. For more information, query on the following words:

   prod(winsdk) and dialog and editor

Additional reference words: 3.00

KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Using UnregisterClass When Removing Custom Control Class

Article ID: Q67248

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

In the Microsoft Windows environment, when no running application
requires a custom control class, the class should be removed from
memory to free the system resources it uses.

If an application registers a control class for temporary use, the
application should use the UnregisterClass function when the control
is no longer needed. If the application is terminated, Windows
automatically removes any classes that the application registered;
therefore, explicit use of UnregisterClass is not required. However,
pairing calls to the RegisterClass and UnregisterClass functions is a
good programming practice.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Using Window Extra Bytes in Custom Controls

Article ID: Q67249

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows environment, custom controls are often designed to store data about their state, color, or position. This data can be stored directly in window extra bytes associated with the custom control window or in a local or a global memory object.

If a control uses window extra bytes directly, Windows allocates memory from the user heap, which is a scarce system resource shared by all applications. In contrast, the LocalAlloc function allocates memory from the heap of the application or the dynamic-link library (DLL) in which the control is implemented.

If six or fewer bytes of data are required for each instance of a custom control, the demand placed on the system user heap is not large. In contrast, each window in Windows version 3.0 requires at least 62 bytes of user heap; therefore, using a small number of window extra bytes is easily justified.

If the control requires more than six bytes of data, the control class should set the number of window extra bytes to the size of one HANDLE data type. When the control is created, call the LocalAlloc function to allocate enough memory to hold the data associated with the control. Store the handle that LocalAlloc returns in the window extra bytes.

More Information:

Depending on the amount of storage required, storing a handle to a local memory block in the window extra bytes is usually advantageous. This method requires calling the LocalAlloc function once at initialization time and calling the LocalLock and LocalUnlock functions each time the control processes a message that refers to or modifies any status information. Calling LocalLock and LocalUnlock is much faster than calling GetWindowWord to access each item of data.

Storing a memory handle in the window extra bytes also provides the control the option of using a global memory block if a very large amount of data must be stored.

Using an allocated memory also allows the application to vary the amount of data stored for each custom control by storing the size of the structure or an index that describes the structure at the beginning of the block of memory.

Additional reference words: 3.00 3.10

KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Generic Custom Control Sample Code in Software Library
Article ID: Q67250

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Software/Data Library contains an archive file CUSTCONT that contains the source code for a generic custom control. The following basic custom control features are demonstrated:

Registering the control class
Creating the control
Painting the control
Changing the font in the control
Gaining and losing focus
Moving and sizing the control

The control is essentially a three-dimensional static rectangle with text. The functions necessary to interact with the Windows Dialog Editor are implemented in a separate module. These functions are documented in Chapter 20 of the "Microsoft Windows Software Development Kit Guide to Programming" for version 3.0.

CUSTCONT contains two archive files. The first, CTL.ZIP, contains the sources for the custom control DLL. The second, CTLTEST.ZIP, contains the sources for a small program that demonstrates the use of the control.

CUSTCONT can be found in the Software/Data Library by searching on the keyword CUSTCONT, the Q number of this article, or S12834. CUSTCONT was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Some CTRL Accelerator Keys Conflict with Edit Controls
Article ID: Q67293

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

SUMMARY
=======

Some keys produce the same ASCII values as CTRL+key combinations. These keys conflict with edit controls if one of the CTRL+key combinations is used as a keyboard accelerator.

The following table lists some of the conflicting keys.

ASCII Value	Key Combination	Equivalent	Windows Virtual Key
0x08	CTRL+H	BACKSPACE	VK_BACK
0x09	CTRL+I	TAB	VK_TAB
0x0D	CTRL+M	RETURN	VK_RETURN

For example, consider the following scenario:

1. CTRL+H has been assigned as an accelerator keystroke to invoke Help

2. An edit control has the focus

3. BACKSPACE is pressed to erase the previous character in the edit control

This results in Help being invoked because pressing BACKSPACE is equivalent to pressing CTRL+H. The edit control does not receive the BACKSPACE key press that it requires because TranslateAccelerator() encounters the 0x08 ASCII value and invokes the action assigned to that accelerator. This limitation is caused by the use of the ASCII key code for accelerators instead of the system-dependent virtual key code.

MORE INFORMATION
================

When messages for the edit control are processed in a message loop that translates accelerators, this translation conflict will occur. Child windows and modeless dialog boxes are the most common situations where this happens.

The affected keystrokes are translated during the processing of the WM_KEYDOWN message for the letter. For example, when the user types CTRL+H, a WM_KEYDOWN is processed for the CTRL key, then another WM_KEYDOWN is processed for the letter "H". In response to this

message, TranslateAccelerator() posts a WM_COMMAND message to the owner of the CTRL+H accelerator. Similarly, when the user presses the BACKSPACE key, a WM_KEYDOWN is generated with VK_BACK as the key code. Because the ASCII value of BACKSPACE is the same as that for CTRL+H, TranslateAccelerator() treats them as the same character. Either sequence will cause a WM_COMMAND message to be sent to the owner of the CTRL+H accelerator, which deprives the child window with the input focus of the BACKSPACE key message.

Because this conflict is inherent to ASCII, the safest way to avoid the difficulty is to avoid using the conflicting sequences as accelerators. Any other ways around the problem may be version dependent rather than a permanent fix.

A second way around the situation is to subclass each edit control that is affected. In the subclass procedure, watch for the desired key sequence(s). The following code sample demonstrates this procedure:

```
/* This code subclasses a child window edit control to allow it to
 * process the RETURN and BACKSPACE keys without interfering with the
 * parent window's reception of WM_COMMAND messages for its CTRL+H
 * and CTRL+M accelerator keys.
 */

/* forward declaration */
long FAR PASCAL NewEditProc(HWND, unsigned, WORD, LONG);

/* required global variables */
FARPROC lpfnOldEditProc;
HWND    hWndOwner;


/* edit control creation in MainWndProc */

TEXTMETRIC tm;
HDC hDC;
HWND hWndEdit;
FARPROC lpProcEdit;

...

 case WM_CREATE:

    hDC = GetDC(hWnd);
    GetTextMetrics(hDC, &tm);
    ReleaseDC(hWnd, hDC);

    hWndEdit = CreateWindow("Edit", NULL,
        WS_CHILD | WS_VISIBLE | ES_LEFT | WS_BORDER,
        50, 50, 50 * tm.tmAveCharWidth, 1.5 * tm.tmHeight,
        hWnd, 1, hInst, NULL);

    lpfnOldEditProc = (FARPROC) GetWindowLong (hWndEdit, GWL_WNDPROC);
    lpProcEdit = MakeProcInstance ((FARPROC) NewEditProc, hInst);
    SetWindowLong(hWndEdit, GWL_WNDPROC, (LONG) lpProcEdit);
    break;
```

```
...

/* subclass procedure */

long FAR PASCAL NewEditProc(HWND hWndEditCtrl, unsigned iMessage,
                            WORD wParam, LONG lParam )
{
  MSG  msg;

  switch (iMessage)
    {
  case WM_KEYDOWN:
    switch (wParam)
        {
    case VK_BACK:
        // This assumes that the next message in the queue will be a
        // WM_COMMAND for the window which owns the accelerators. If
        // this edit control were in a modeless dialog box, hWndOwner
        // should be set to NULL. It may also be NULL in this case.
        PeekMessage(&msg, hWndOwner, 0, 0, PM_REMOVE);

        // Since TranslateAccelerator() processed this message as an
        // accelerator, a WM_CHAR message must be supplied manually to
        // the edit control.
        SendMessage(hWndEditCtrl, WM_CHAR, wParam, MAKELONG(1, 14));
        return 0L;

    case VK_RETURN:
        // Same procedures here.
        PeekMessage(&msg, hWndOwner, 0, 0, PM_REMOVE);
        SendMessage(hWndEditCtrl, WM_CHAR, wParam, MAKELONG(1, 28));
        return 0L;
        }
        break;
    }
    return CallWindowProc(lpfnOldEditProc, hWndEditCtrl, iMessage,
                          wParam, lParam);
}
```

NOTE: Be sure to export the subclass function in the DEF file.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlEdit

PRB: Documented EM_SETWORDBREAK Message Does Nothing

Article ID: Q67611

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Page 6-29 of the "Windows Software Development Kit Reference Volume 1" version 3.0 documents the EM_SETWORDBREAK message for multiline edit controls. This message is designed to specify an application-supplied word-break function. This function is called by a multiline edit control when the edit control contents will not fit on a single line. The function supplies the point at which Windows should break the line.

However, this message has no effect on Windows version 3.00 edit controls. Windows will always use its default word-break procedure, which breaks a line at a blank character.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Captions for Dialog List Boxes
Article ID: Q24646

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To place text into the caption bar specified for a list box, use the SetWindowText() function. First, use GetDlgItem() to get the handle of the list box, then call SetWindowText() to set the list box caption.

The following code fragment from the Windows Software Development Kit (SDK) TEMPLATE application illustrates the necessary steps (please note that TEMPLATE.RC was modified so the list box would include the WS_CAPTION window style):

```
   ...
   BOOL FAR PASCAL TemplateDlg(hWndDlg, message, wParam, lParam)
   ...
       switch (message)
        {
       case WM_INITDIALOG:
           ...
           /* The following line sets the Listbox caption */
           SetWindowText( GetDlgItem(hWndDlg, IDDLISTBOX), (LPSTR)"Caption");
           for (i = 0; i < CSTR; i++) {
               LoadString(hInstTemplate, IDSSTR1+i, (LPSTR)szWaters, 12);
               SendDlgItemMessage(hWndDlg, IDDLISTBOX, LB_ADDSTRING, 0,
                   (LONG)(LPSTR)szWaters);
               }
           SendDlgItemMessage(hWndDlg, IDDLISTBOX, LB_SETCURSEL, iSel, 0L);
           ...
           return TRUE;

       case WM_COMMAND:
   ...
```

Additional reference words: 2.00 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Multiline Edit Control Overwrite Mode Sample Code
Article ID: Q81610

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

ECOVRWRT is a file in the Software/Data Library that demonstrates how an application can implement a multiline edit control in which the user can toggle between "insert" mode and "overwrite" mode. In insert mode, new text pushes any existing text further to the right. In overwrite mode, new text replaces any existing text.

To implement overwrite mode, ECOVRWRT subclasses the multiline edit control. When the control is in overwrite mode, when the control receives a WM_CHAR message, the character after the selection point is selected and replaced with the character in the wParam of the message.

ECOVRWRT can be found in the Software/Data Library by searching on the word ECOVRWRT, the Q number of this article, or S13304. ECOVRWRT was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib ECOVRWRT.ZIP
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Changing/Setting the Default Push Button in a Dialog Box
Article ID: Q67655

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The default push button in a dialog box is defined to be the button that is pressed when the user presses the ENTER key, provided that the input focus is not on another button in the dialog box. This button is visually distinguished from other buttons by a thick dark border. This article describes how to change the default push button.

More Information:

To change the default push button, perform the following three steps:

1. Send the BM_SETSTYLE message to the current default push button to change its border to that of a regular push button.

2. Send a DM_SETDEFID to the dialog box to change the ID of the default push button.

3. Send the BM_SETSTYLE message to the new default push button to change its border to that of a default push button.

The following is sample code that performs the three steps:

```
   // Reset the current default push button to a regular button.
   SendDlgItemMessage(hDlg, <ID of current default push button>,
         BM_SETSTYLE, BS_PUSHBUTTON, (LONG)TRUE);

   // Update the default push button's control ID.
   SendMessage(hDlg, DM_SETDEFID, <ID of new default push button>,
         0L);

   // Set the new style.
   SendDlgItemMessage(hDlg, <ID of new default push button>,
         BM_SETSTYLE, BS_DEFPUSHBUTTON, (LONG)TRUE);
```

Note, however, that ANY push button that has the input focus will have a dark border. A default push button will retain this dark border even when the input focus is transferred to another control in the dialog box, provided the new control is not another push button.

For example, if the input focus is on an edit control, check box, radio button, or any control other than a push button, and the ENTER key is pressed, Windows sends a WM_COMMAND message to the dialog box procedure with the wParam set to the control ID of the default push button.

Additional reference words: 3.00 MICS3 R5.7
KBCategory:
KBSubcategory: UsrCtlButtons

INF: Sample Code Demonstrates an Owner-Draw Combo Box

Article ID: Q81706

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When Windows creates an owner-draw combo box, it sends WM_DRAWITEM messages to the parent window of the combo box. To respond to this message, the parent window must perform the steps necessary to draw the items in the owner-draw combo box.

OWNCOMBO is a sample in the Software/Data Library that contains an example of a fixed-height owner-draw combo box that contains strings. The combo box also demonstrates painting bitmaps, changing the text color, and storing data associated with each item using the CB_SETITEMDATA message. All of the relevant code is contained within the ODDLG.C module.

OWNCOMBO can be found in the Software/Data Library by searching on the word OWNCOMBO, the Q number of this article, or S13305. OWNCOMBO was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib OWNCOMBO.ZIP
KBCategory:
KBSubcategory: UsrCtlOdcombo

INF: WM_CTLCOLOR Processing for Combo Boxes of all Styles
Article ID: Q81707

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Windows sends a WM_CTLCOLOR message to the parent of a control window to enable the parent to specify the color of the control. A notification code, which is the value of the high-order word of the lParam, accompanies the WM_CTLCOLOR message to indicate the control type for a WM_CTLCOLOR message. Windows does not define a notification code that enables an application to change the color of a combo box control. However, Windows sends WM_CTLCOLOR messages to a combo box control that relate to its component parts: one message for the list box portion and, if applicable, another message for the edit control portion. An application can subclass the control to intercept and process these messages. This article discusses how to perform the subclassing and how to address the problems that arise when an application changes the colors of a combo box under Windows 3.0.

More Information:

Under Windows versions 3.0 and later, Windows sends a WM_CTLCOLOR message to a combo box for each of its individual elements. An application can subclass the combo box control to process the message. The text below discusses each of the combo box styles and how the color can be changed for each style.

CBS_SIMPLE Style

Under Windows 3.0, an application must process three WM_CTLCOLOR notifications codes, CTLCOLOR_EDIT, CTLCOLOR_MSGBOX, and CTLCOLOR_LISTBOX, to change the colors for a CBS_SIMPLE-style list box. Under Windows 3.1, the application is not required to process the CTLCOLOR_MSGBOX notification.

Each time the application processes a notification, it must set the foreground and background colors, using SetTextColor and SetBkColor, respectively. The wParam accompanying the WM_CTLCOLOR message contains a handle to the appropriate display context. In addition, the application must return a valid handle to the appropriate background brush that Windows will use to paint those areas not occupied by text.

CBS_DROPDOWN Style

To change the colors of a CBS_DROPDOWN-style combo box under Windows 3.0, process the same three notifications as for the CBS_SIMPLE combo box. However, there is a difficulty with regard to setting the text

color. In the drop-down list box, the colors set with SetBkColor and SetTextColor are not used. Instead, the combo box uses the system default colors. As a workaround, change the combo box to the owner-draw style. Process the WM_DRAWITEM message to draw the individual items with the desired text colors.

For more information on using an owner-draw combo box in an application, query on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and owncombo

Under Windows 3.1, it is not necessary to use an owner-draw combo box. The application can process the CTLCOLOR_EDIT and CTLCOLOR_LISTBOX notifications to change the color of a combo box.

CBS_DROPDOWNLIST Style

To change the color of a CBS_DROPDOWNLIST-style combo box, process the CTLCOLOR_LISTBOX notification. However, the application must process this notification in the combo box subclass procedure and in the window procedure for the parent window of the combo box.

Under Windows 3.0, the text color problem discussed above for CBS_DROPDOWN-style combo boxes is evident for CBS_DROPDOWNLIST combo boxes. To address this problem, the application must use an owner-draw combo box.

Although it is not necessary to use an owner-draw combo box under Windows 3.1, the application must process the CTLCOLOR_LISTBOX notification in both the combo box subclass procedure and in the window procedure for the parent window of the combo box.

Additional reference words: 3.00 3.10 combobox listbox
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Controlling the Horizontal Scroll Bar on a List Box

Article ID: Q67677

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Software Library contains an archive of a document that explains how to manipulate the horizontal scroll bar on a Windows version 3.0 list box. LBOXSBAR can be found in the Software/Data Library by searching on the word LBOXSBAR, the Q number of this article, or S12852. LBOXSBAR was archived using the PKware file-compression utility.

LBOXSBAR refers to two other Software Library files that contain sample code: LISTHORZ and LISTHSCR. Each archive contains code to maintain a horizontal scroll bar on a Windows list box. The code in LISTHORZ is in a C module that can be compiled and linked to any Windows application. The code in LISTHSCR is in a Windows DLL that can support multiple list boxes in the same application.

LISTHORZ can be found in the Software/Data Library by searching on the word LISTHORZ, Q66370, or S12804. LISTHSCR can be found in the Software/Data Library by searching on the word LISTHSCR, Q66370, or S13558. LISTHORZ and LISTHSCR were archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: List Box Capacity Limits

Article ID: Q67678

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 2.0, 2.03, 2.10, 3.0 and 3.1

SUMMARY
=======

In the Microsoft Windows graphical environment, the storage capacity of a list box is limited by the total amount of text. Each item is stored in memory as a NULL-terminated string.

The maximum amount of memory that a list box can use to store text depends on the version of Windows in use. In Windows versions 2.x, the amount of free memory in the user module's data segment limits available storage.

Beginning with Windows 3.0, each list box has its own data segment, which allows it to store 64K of text. If a list box in an application must store more text, the application can store up to 8160 items in an owner-draw list box that does not have the LBS_HASSTRINGS list box style. In this case, the application stores and manages the item text separately.

MORE INFORMATION
================

An owner-draw list box is limited to 64K of data storage, as are regular (nonowner draw) list boxes. Because the application manages the text when the list box does not have the LBS_HASSTRINGS style, each item in an owner-draw list box requires 8 bytes, 4 of which store the application-supplied 32-bit item-identifier value.

The list box uses 256 bytes of the 64K segment for internal storage, and uses the rest for items. This yields room for 8160 items, as follows:

```
   (64K segment - 256 bytes) / (8 bytes per item)
```

NOTE: In a list box with the LBS_HASSTRINGS style, the text pointed to by the pointer passed in the lParam of LB_ADDSTRING (or LB_INSERTSTRING) is stored in the list box's storage area. An additional 32-bit storage value can be associated with each item with the LB_SETITEMDATA style.

When the LBS_HASSTRINGS style is not present, Windows no longer stores text for a list box. Instead of storing the text pointed to by the lParam of LB_ADDSTRING (or LB_INSERTSTRING), only the 32-bit value itself is stored, thus dramatically reducing the needed storage. When the owner-draw list box receives the WM_DRAWITEM message, it is up to the drawing routine to interpret the 32-bit value.

Additional reference words: 2.00 2.03 2.10 3.00 3.10 listbox
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Sample Code Implements a

Article ID: Q81814

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The Image Editor and the Dialog Editor are two applications provided with the Windows Software Development Kit (SDK) that demonstrate a "tool box" window. This article discusses how to implement a tool box window in an application. TOOLBOX is a file in the Software/Data Library that demonstrates the techniques discussed.

TOOLBOX can be found in the Software/Data Library by searching on the word TOOLBOX, the Q number of this article, or S13308. TOOLBOX was archived using the PKware file-compression utility.

More Information:

A tool box window contains a group of icons that the user can choose at any time. The tool box window always floats above the application with which it is associated.

The code in TOOLBOX assumes that all buttons in the tool box have the same size and are equally spaced throughout the window. TOOLBOX implements the following four steps to create the tool box window:

1. Loads the bitmap that provides the visual appearance for all the buttons.

2. Creates the tool box window with the WS_EX_TOPMOST style bit set. Sizes the window appropriately based on the size of the bitmap.

3. Processes the WM_ACTIVATE message in the window procedure for the application's main window. If the main window or the tool box window is being activated, TOOLBOX calls SetWindowPos to add the WS_EX_TOPMOST style to the window. If a window other than the main window or the tool box window is being activated, TOOLBOX calls SetWindowPos to remove the WS_EX_TOPMOST style from the window.

4. Performs the following two steps in the window procedure for the tool box window:

   a. Processes the WM_LBUTTONDOWN message. Performs hit testing to determine which button the user chose. To make the button appear depressed, TOOLBOX determines the rectangle for the button. TOOLBOX uses the InvertRect function to invert the colors of the button.

   b. Processes the WM_PAINT message. TOOLBOX uses the StretchBlt function to paint the bitmap onto the window. If any button is

depressed, TOOLBOX inverts the appropriate region as above.

Although this sample does not demonstrate every possibility for tool box windows, the sample is very straightforward and easily modified.

Additional reference words: 3.10 softlib TOOLBOX.ZIP
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Sample Program Demonstrates Edit Control Validation
Article ID: Q82076

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

DATAVAL is a file in the Software/Data Library that demonstrates how to validate the contents of one or more edit controls within the same dialog box without subclassing the edit controls.

DATAVAL can be found in the Software/Data Library by searching on the word DATAVAL, the Q number of this article, or S13369. DATAVAL was archived using the PKware file-compression utility.

More Information:

DATAVAL processes the EN_KILLFOCUS message, which the dialog box receives after an edit control has lost the input focus. When the dialog box receives an EN_KILLFOCUS message, it calls a validation function to determine if the contents of the control losing focus is valid. If the contents is not valid, then the dialog box posts a user-defined "not valid" message to itself. When the dialog box processes the "not valid" message, it displays a message box alerting the user of the error, and then sets focus back to the control with invalid contents. A cancel button in the dialog box allows the user to exit the dialog box without performing validation.

An application is in an unstable state when it receives an EN_KILLFOCUS message; Windows is in the process of switching focus between windows. Therefore, the application must use PostMessage to provide the "not valid" message or the application can enter an infinite loop.

Additional reference words: 3.00 softlib DATAVAL.ZIP
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Combo Box w/Edit Control & Owner-Draw Style Incompatible

Article ID: Q82078

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The owner-draw combo box styles (CBS_OWNERDRAWFIXED and CBS_OWNERDRAWVARIABLE) are incompatible with the combo box styles that contain an edit control (combo box styles CBS_SIMPLE and CBS_DROPDOWN). A combo box with either the CBS_SIMPLE or CBS_DROPDOWN style displays the currently selected item in its associated edit control. When an owner-draw style is specified for the combo box style CBS_SIMPLE or CBS_DROPDOWN, the current selection may not be displayed. Using the SetWindowText function to display the current selection in response to a CBN_SELCHANGE message may not be effective.

More Information:

An owner-draw combo box can contain bitmaps or other graphic elements in its list box. Therefore, to correctly display the current selection, it is necessary to display a bitmap or other graphic element in the edit control. Because edit controls are not designed to display graphics, there is no natural method to display the current selection in an owner-draw combo box with an edit control.

The combo box style CBS_DROPDOWNLIST, which has a static text area instead of an edit control, can display any item, including graphics. Use this style combo box with the owner-draw styles.

Additional reference words: 3.00 3.10 3.x SR# G920226-89
KBCategory:
KBSubcategory: UsrCtlOdcombo

INF: Owner-Draw Buttons with Bitmaps on Non-Standard Displays

Article ID: Q67715

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

If an application contains an owner-draw button that paints itself with a bitmap, the application's resources must contain a set of bitmaps appropriate to each display type on which the application might run.

If the application's resources do not contain bitmaps suitable for the display on which the application is running, the application can use the default 3-D button appearance by changing the button style to BS_PUSHBUTTON from BS_OWNERDRAW.

Changing the style of a button is possible in Windows version 3.0; however, this technique is not guaranteed to be supported in future releases of Windows.

For more information on owner-draw controls, please query on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and owner-draw

More Information:

An owner-draw button can use bitmaps to paint itself. When an application contains this type of owner-draw button, it must also contain a set of bitmaps appropriate for each display type on which the application might run. Each set of bitmaps has a normal "up" bitmap and a depressed "down" bitmap to implement the 3-D effects. The most common standard Windows display types are: CGA, EGA, VGA, 8514/a, and Hercules Monochrome. The dimensions and aspect ratio of the display affect the appearance of the bitmap. For example, a monochrome bitmap designed for VGA will display correctly on an 8514/a and any other display with a 1:1 aspect ratio.

If an application determines that it does not contain an appropriate set of bitmaps for the current display type, then it should change the button style from BS_OWNERDRAW to BS_PUSHBUTTON. After the style has been changed and the button has been redrawn, the button will appear as a normal 3-D push button.

The following code fragment demonstrates how to change the style of a push button from owner-draw to normal:

```
    ...

    /*
     * hWndButton is assumed to be the handle to the button.
```

```
     * Note that lParam has a nonzero value, which forces the button
     * to be redrawn. This assures that the normal button appearance
     * will show after this message is sent.
     */
    SendMessage(hWnd, BM_SETSTYLE, BS_PUSHBUTTON, 1L);

    ...
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlOdbuttons

INF: Assigning Mnemonics to Owner-Draw Push Buttons

Article ID: Q67716

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.0

An application that uses owner-draw push buttons is always responsible for the appearance of the buttons. It might seem that in doing so, the ability to assign a mnemonic character to an owner-draw button is lost because text containing the mnemonic may not be displayed.

Fortunately, this is not the case. If an owner-draw button should be activated by ALT+X, place "&X" into the button text.

When the ALT key is pressed in combination with any character, Windows examines the text of each control to determine which control, if any, uses that particular mnemonic. With an owner-draw button, the text exists, but may not necessarily be used to paint the button.

For more information on owner-draw buttons, query on the following word:

owner-draw

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlOdbuttons

INF: Multiline Edit Control Wraps Text Different than DrawText

Article ID: Q67722

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows,
   version 3.0

SUMMARY
=======

Multiline edit controls will not wrap text in the same manner as the DrawText() function. This can be a problem when an application displays text that has been in an edit control because the text may wrap in a different location.

It is possible to obtain the text from the edit control and display it statically in a window with the same line breaks. To do this, the application must retrieve each line of text separately. This can be accomplished by sending the EM_GETLINE message to the control and displaying the retrieved text with the TextOut() function.

MORE INFORMATION
================

The following is a brief code fragment that demonstrates how to obtain the text of a multiline edit control line by line:

```
   ... /* other code */

   char  buf[80];          // Buffer for line storage
   HDC   hDC;              // Temporary display context
   HFONT hFont;            // Temporary font storage
   int   iNumEditLines;    // How much text
   TEXTMETRIC tm;          // Text metrics

   // Get number of lines in the edit control
   iNumEditLines = SendMessage(hEditCtl, EM_GETLINECOUNT, 0, 0L);

   hDC = GetDC(hWnd);

   // Get font currently selected into the control
   hFont = SendMessage(hEditCtl, WM_GETFONT, 0, 0L);

   // If it is not the system font, then select it into DC
   if (hFont)
      SelectObject(hDC, hFont);

   GetTextMetrics(hDC, &tm);
   iLine = 0;

   while (iNumEditLines--)
      {
```

```
      // First word of buffer contains max number of characters
      // to be copied
      buf[0] = 80;

      // Get the current line of text
      nCount = SendMessage(hEditCtl, EM_GETLINE, iLine, (LONG)buf);
      TextOut(hDC, x, y, buf, nCount);  // Output text to device
      y += tm.tmHeight;
      iLine++;
      }

   ReleaseDC(hWnd, hDC);
   ... /* other code */
```

The execution time of this code could be reduced by using the ExtTextOut() function instead of TextOut().

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

BUG: Single-Line Edit Controls and Fonts

Article ID: Q67739

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012018

SYMPTOMS
   When using a single-line edit control with a font larger than the
   system default font, deleting characters with the backspace key
   causes painting problems. Specifically, the lower half of a deleted
   character is not removed from the screen.

RESOLUTION
   This problem can be avoided by using a multiline edit control sized
   to hold only one line. NOTE: Do not specify the ES_AUTOVSCROLL
   style, or the multiline edit control will accept more than one line
   of text.

   Microsoft has confirmed this to be a problem in Windows version
   3.0. We are researching this problem and will post new information
   here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Trapping the INS and DEL Keys
Article ID: Q19751

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS
   When running the TRACK sample application application, a message box appears verifying that a key was pressed. This works for most keys except INS and DEL. However, the DLGC_WANTALLKEYS code would not trap the INS and DEL keys as expected.

   TRACK.C can be found in the Software/Data Library by searching on the keyword TRACK, the Q number of this article, or S12038. TRACK was archived using the PKware file-compression utility.

CAUSE
   A list box does not use the INS or DEL keys. If you want to trap the VK_DELETE and VK_INSERT keys, you must add a switch statement to the List Box window procedure.

RESOLUTION
   You must add a switch statement to the List Box procedure for VK_DELETE and VK_INSERT.

More Information:

The following is the list box procedure from TRACK.C with a switch statement added for VK_DELETE and VK_INSERT:

```
/********************************************************************
 * This is the new window procedure for the ListBox control.        *
 ********************************************************************/
long FAR PASCAL NewListWndProc(hwnd, message, wParam, lParam)
HWND hwnd;
unsigned message;
WORD wParam;
LONG lParam;
{
   char  temp[80];   /* added */

   switch (message) {
     case WM_GETDLGCODE:
          /**********************************************************
           *  When the parent needs to know what keys the control    *
           *  needs to process, it sends this message and it is      *
           *  up to the control to let it know.  In this case        *
           *  here, the control would like all the keys.             *
           **********************************************************/
          return (DLGC_WANTALLKEYS);
```

```
        break;
     case WM_KEYDOWN:
          switch(wParam)
           {
           case VK_DELETE:
               MessageBox(hwnd,(LPSTR)"Del",(LPSTR)"hit",MB_OK);
               break;

           case VK_INSERT:
               MessageBox(hwnd,(LPSTR)"Ins",(LPSTR)"hit",MB_OK);
               break;
           }
        break;
     case WM_CHAR:
          /**********************************************************
           *  Warning: Make sure that when you intercept messages    *
           *  and then pass them on, make sure they are going to the *
           *  correct Window, or in some cases to the PARENT.        *
           *  A case of this may be where you want to convert a ENTER*
           *  into a TAB; you will want to pass the converted TAB    *
           *  to the PARENT or the HANDLE to the DIALOG in this case.*
           **********************************************************/
          sprintf(temp,"%x was pressed",wParam);
          MessageBox(hwnd, (LPSTR)temp, (LPSTR)"WM_CHAR Hit", MB_OK);
          if (wParam == VK_TAB ||wParam == VK_RETURN) {
            /* Let's send a message to the parent to let it know
               what's going on  */
            SendMessage (hParent, WM_USER,0,0L);
            SetActiveWindow (hPushButton); /* put focus on next control */
            return TRUE;
          }
          else
            return CallWindowProc(lpfnListOld, hwnd, message, wParam, lParam);
        break;
     default:
          /*********************************************************
           *  This returns all the unprocessed messages back to     *
           *  the original procedure.                               *
           *********************************************************/
          return CallWindowProc(lpfnListOld, hwnd, message, wParam, lParam);
    } /* end switch */
}
```

Additional reference words: 2.x 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Creating a List Box with No Vertical Scroll Bar
Article ID: Q68115

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A Windows list box with the LBS_STANDARD style will display a vertical scroll bar if there are more items in the list than can be displayed in the client area of the list box.

To create a list box that will not use a vertical scroll bar, you must remove the WS_VSCROLL bit from the window style. The following style specification removes this bit:

```
   (LBS_STANDARD | LBS_HASSTRINGS) & ~WS_VSCROLL
```

Additional reference words: 2.00 2.03 2.10 3.00 2.x
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Creating a List Box That Does Not Sort

Article ID: Q68116

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A Windows list box with the LBS_STANDARD style will sort the list of items into alphabetical order before displaying them in the control.

To create a list box that will not sort, you must remove the LBS_SORT bit from the window style. The following style specification removes this bit:

```
   (LBS_STANDARD | LBS_HASSTRINGS) & ~LBS_SORT
```

Additional reference words: 2.00 2.03 2.10 3.00 2.x
KBCategory:
KBSubcategory: UsrCtlListbox

PRB: WM_GETTEXT Documentation Error in SDK Reference Volume 1
Article ID: Q71143

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

On page 6-64 in the "Windows Software Development Kit Reference Volume 1," the description of the message WM_GETTEXT is incorrect. The description of the WM_GETTEXT function incorrectly reads:

> For lixt [sic] boxes, the text is the currently selected item.

RESOLUTION/STATUS

WM_GETTEXT does not copy the selected text for list boxes. To copy the selected text from a list box, the LB_GETTEXT function should be used instead.

Microsoft has confirmed this to be a documentation error on page 6-64 of the "Windows Software Development Kit Reference Volume 1." The description will be corrected in a future version of the Windows SDK documentation.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Developing a Spreadsheet Application for Windows
Article ID: Q68301

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A spreadsheet, such as Microsoft Excel, is used to store and manipulate data. It has a rectangular grid arranged in columns and rows. The intersection of each column and row is a cell, the basic unit of a spreadsheet in which the application stores data.

To develop a spreadsheet application, the predefined control types (such as list boxes and edit controls) should not be used to represent cells. These controls have limitations that preclude using them for large scale purposes.

The information below describes a more efficient alternative to using predefined controls when creating a spreadsheet type application. There is a file in the Software/Data Library named SPDSHEET that contains the sample code to demonstrate such an application.

More Information:

The main technique used to develop a spreadsheet application for Windows is to draw vertical and horizontal lines on a window to represent the cells of a spreadsheet. Mouse hit-testing is then done to track the currently active spreadsheet cell. To enter the spreadsheet data, only one edit control is used. The most recent data is painted simultaneously on the current cell as it is entered by the user. To store the spreadsheet data internally, an array of the desired data type is created.

All the spreadsheet data and other visual information is repainted on the main window whenever the application gets a WM_PAINT message.

The SPDSHEET sample application uses Windows API function calls to paint the spreadsheet cells. It draws all of the solid lines using MoveTo() and LineTo() functions. The vertical dotted lines are drawn by calling LineDDA(), which in turn calls a callback function that draws pixels. The application uses DrawText() to display column letters and row numbers. The highlight on the current cell is achieved by inverting the bits on all four sides of the cell.

Since this is a simplified version of a spreadsheet, the sample application creates an array of 100 strings to represent spreadsheet data. The developers of a spreadsheet application can choose to use their own data type.

The SPDSHEET example also creates two other windows on the upper side of its client area. One is a static window to display the position of

the current selected cell and the other is an edit control that is used to enter spreadsheet data. As the data is entered to the edit control, it is also drawn to the current selected cell using DrawText().

Upon receiving a WM_LBUTTONDOWN message, the application removes the highlight from the currently selected cell and calculates the location of the newly selected cell. The application highlights the newly selected cell and copies the cell's contents to the edit control. The static window is also updated to display the location of the newly selected cell.

The information corresponding to each cell's location is not stored by SPDSHEET because the cell size is constant and the location can be calculated. If, on the other hand, the cells were of variable sizes, some location information would have to be stored and managed by the application.

When the application receives a WM_PAINT message, it repaints its entire client area, including the whole entire array of spreadsheet data. To make this process more efficient, the application can make use of the invalid region information sent by Windows through the PAINTSTRUCT structure.

SPDSHEET can be found in the Software/Data Library by searching on the word SPDSHEET, the Q number of this article, or S12879. SPDSHEET was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Changing a List Box from Single-Column to Multicolumn

Article ID: Q68580

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

A list box cannot be changed from single column to multicolumn by altering the list box's style bits "on the fly."

The effect of switching a single-column list box to multicolumn can be achieved by creating one single column and one multicolumn list box. Initially, hide the multicolumn list box. To switch the list boxes, hide the single-column list box and show the multicolumn list box.

More Information:

In general, programmatically changing the style bits of a window usually leads to unstable results. The method of switching between two windows (hiding one and showing the other) can safely switch window styles.

Additional reference words: 3.00 3.10 listbox
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Subclassing Warning in Windows SDK Reference Volume 1

Article ID: Q68584

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Window subclassing is explained on page 1-17 of the "Microsoft Windows Software Development Kit Reference Volume 1," in the section titled "Window Subclassing."

At the end of that section, the following warning appears:

Note: An application should not attempt to create a window subclass for standard Windows controls such as combo boxes and buttons.

This warning means that SetClassLong() should not be used to subclass an entire standard control class. This would cause all controls of that type (including controls in other applications) created while the subclass was in effect to be subclassed.

SetWindowLong() can be used to subclass individual controls in your application.

In addition, standard Windows controls should only be subclassed in "non-intrusive" ways. Subclassing procedures that alter the appearance of a control or that depend on undocumented messages or message parameters could be incompatible with future versions of Windows.

More Information:

For more information on subclassing controls, query on the following word:

subclassing

Charles Petzold's "Programming Windows" (Microsoft Press) contains a sample program that shows how to subclass a standard control (in this case, a scroll bar). In the Windows 2.x version of the book, the sample is called COLORSCR; in the Windows 3.00 version, the sample is called COLORS1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlSubclass

INF: How to Simulate Changing the Font in a Message Box
Article ID: Q68586

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To simulate changing the font in a message box, create a dialog box that uses the desired font. Specify the style and contents of the dialog box to reflect the style of the desired message box. The application can also draw a system icon in the dialog box.

More Information:

The message box is a unique object in Windows. Its handle is not available to an application; therefore, it cannot be modified. An application can simulate a message box with a different font by creating a dialog box that looks like a message box.

To change the font in a dialog box, use the optional statement FONT in the dialog statement of the resource script (.RC) file. For example, resource file statements for a dialog box displaying an error in Courier point size 12 would be as follows:

```
   FontError DIALOG  45, 17, 143, 46
   CAPTION "Font Error"
   FONT 12, "Courier"
   STYLE WS_CAPTION | WS_SYSMENU | DS_MODALFRAME
   BEGIN
       CTEXT "Please select the right font", -1, 0, 7, 143, 9
       DEFPUSHBUTTON "OK"  IDOK,   56, 25,  32, 14, WS_GROUP
   END
```

To center the dialog box in the screen, use GetWindowRect() to retrieve the dimensions of the screen and MoveWindow() to place the dialog box appropriately. The following code demonstrates this procedure:

```
   case WM_INITDIALOG:
       GetWindowRect(hDlg, &rc);
       x = GetSystemMetrics(SM_CXSCREEN);
       y = GetSystemMetrics(SM_CYSCREEN);
       MoveWindow (hDlg,
          (x - (rc.right - rc.left)) >> 1,  /* x position */
          (y - (rc.bottom - rc.top)) >> 1,  /* y position */
          rc.right - rc.left,               /* x size */
          rc.bottom - rc.top,               /* y size */
          TRUE);                            /* paint the window */
       return TRUE;
```

To display a system icon in the dialog box, call the DrawIcon() function during the processing of a WM_PAINT message. After drawing

the desired icon, the dialog procedure passes control back to the dialog manager by returning FALSE. The code to paint the exclamation point icon (used in warning messages) is as follows:

```
   case WM_PAINT:
       hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
       hDC = GetDC(hDlg);
       DrawIcon(hDC, 20, 40, hIcon);
       ReleaseDC(hDlg, hDC);
       return FALSE;
```

Additional reference words: 3.00 3.0
KBCategory:
KBSubcategory: UsrCtlSetfont

INF: Handling WM_CANCELMODE in a Custom Control

Article ID: Q74548

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

In the Microsoft Windows graphical environment, the WM_CANCELMODE message informs a window that it should cancel any internal state. This message is sent to the window with the focus when a dialog box or a message box is displayed, giving the window the opportunity to cancel states such as mouse capture.

When a control has the focus, it receives a WM_CANCELMODE message when the EnableWindow function disables the control or when a dialog box or a message box is displayed. When a control receives this message, it should cancel modes, such as mouse capture, and delete any timers it has created. A control must cancel these modes because an application may use a notification from the control to display a dialog box or a message box.

The DefWindowProc function processes WM_CANCELMODE by calling the ReleaseCapture function, which cancels the mouse capture for whatever window has the capture. The DefWindowProc function does not cancel any other modes.

More Information:

For example, consider a miniature scroll bar custom control that, when it receives a mouse click, sets the mouse capture, creates a timer to provide for repeated scrolling, and sends a WM_VSCROLL message to its parent application. The timer is used to send WM_VSCROLL messages periodically to the parent when the mouse button is held down and the mouse is over the control.

If the application displays a dialog box in response to the WM_VSCROLL message, the control receives a WM_CANCELMODE message, at which time it should kill its timer and release the mouse capture. If the WM_CANCELMODE message is simply passed to the DefWindowProc function, only the mouse capture is released; the timer remains active. When the dialog box is closed, the control immediately sends the parent another WM_VSCROLL message, causing it to display the dialog box again.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Location of the Cursor in a List Box

Article ID: Q29961

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There is no way to determine which item in the list box has the cursor
when the LBN_DBLCLK message is received. You must keep track of which
item has the cursor as it moves among the items. When you receive the
double-click message, you will know which box has the cursor.

Additional reference words: 2.03 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Simulating the Drag-and-Drop Interface for Custom Control
Article ID: Q69080

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Software/Data Library contains a file named MOVECC, which demonstrates how to simulate the Drag and Drop interface for custom controls. By using the MoveWindow() function, this sample allows you to click on a control and drag it to a new location.

MOVECC also demonstrates how to manipulate child window Z-order to prevent windows from going behind other child windows, and to avoid children painting on other children.

MOVECC can be found in the Software/Data Library by searching on the keyword MOVECC, the Q number of this article, or S12913. MOVECC was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Sample Code to Demonstrate Superclassing Available
Article ID: Q32167

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There is a sample application available in the Software Library that demonstrates the concept of superclassing control windows. Superclassing provides the same capabilities as subclassing; however, it can be much more convenient to use and produces smaller code if there are multiple controls to be altered in the same fashion.

SUPERCLS can be found in the Software/Data Library by searching on the word SUPERCLS, the Q number of this article, or S12007. SUPERCLS was archived using the PKware file-compression utility.

More Information:

To superclass, create a new class that uses the control window procedure instead of the default window procedure. This enables the application to process the appropriate messages and to pass the others along to the default procedure where the messages will be processed correctly based on the control type.

To implement superclassing, create a window of the type to superclass. Query all available information about the control and use this information when the new window is created, substituting the class name and window procedure. When this window procedure is called, process the desired messages and pass the others to the standard control procedure (determined in the query phase).

By registering a new class, the application can create superclassed controls using the application resource (RC) file or CreateWindow() function without any further work, which can eliminate a number of SetWindowLong() calls if many controls are to be modified in the same manner.

In the SUPERCLS example, the application creates a new control class called "SuperEdit", which is identical to a regular edit control but refuses to accept the "b" character.

Look for the string "SuperEdit" in the .C, .RC, and .DEF files to see the affected areas of the code.

This example shows superclassed Windows controls. Because the class name has been changed and the application's instance is used in the class registration, all controls will be created with this application's DS. This causes no problems for controls. However, this does not work if an attempt is made to superclass something other than a control. In that case, it is desirable to use the application's

instance with the class so that it will be destroyed when the
application exits. However, it is necessary to perform some additional
work before the call to CallWindowProc() to set the DS for the
subclassed window procedure. The hInstance passed to CreateWindow()
determines which DS the superclassing procedure uses; to have the
superclassed procedure use a different DS, it is necessary to change
the DS value. For more information, see the notes near the call to
CallWindowProc().

Each edit control created using DialogBox(), CreateDialog(), or
through an RC file normally has its own DS. When the application
changes the class name, the code within Windows that handles edit
controls is not activated. Instead, the control uses the application's
DS as if it were created using CreateWindow(). This does not cause any
difficulties except that when the code for the superclassed edit
control calls LocalAlloc(), it consumes space in the application's
local heap. This should not cause problems in most cases.

Additional reference words: 2.x 3.00 sub class subclass control 2.03
2.10 softlib SUPERCLS.ZIP
KBCategory:
KBSubcategory: UsrCtlSuperclass

INF: Placing Text in an Edit Control

Article ID: Q32785

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Text is placed into an edit control by calling SetDlgItemText() or by sending the WM_SETTEXT message to the edit control window, with lParam being a far pointer to a null-terminated string. This message can be sent in two ways:

1. SendMessage(hwndEditControl, WM_SETTEXT, ...

2. SendDlgItemMessage(hwndParent, ID_EDITCTL, WM_SETTEXT...

   Note: hwndParent is the window handle of the parent, which may be a dialog or window. ID_EDITCTL is the ID of the edit control.

Text is retrieved from an edit control by calling GetDlgItemText() or by sending the WM_GETTEXT message to the edit control window, with wParam being the maximum number of bytes to copy and lParam being a far pointer to a buffer to receive the text. This message can be sent in two ways:

1. SendMessage(hwndEditControl, WM_GETTEXT, ...

2. SendDlgItemMessage(hwndParent, ID_EDITCTL, WM_GETTEXT...

   Note: hwndParent is the window handle of the parent, which may be a dialog or window. ID_EDITCTL is the ID of the edit control.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Default Edit Control Entry Validation Done by Windows
Article ID: Q74266

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

Under the Microsoft Windows graphical environment, multiline and single-line edit controls do not accept characters with virtual key code values less than 0x20. The two exceptions are the TAB and ENTER keys; users can enter these characters only in a multiline edit control.

If an application creates an edit control with the ES_LOWERCASE or ES_UPPERCASE style, text entry is converted into the specified case.

If an application creates an edit control with the ES_OEMCONVERT style, the text is converted from the ANSI character set to the OEM character set and then back to ANSI for display in the control.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlValidate

INF: Read-Only Edit Control Sample Compatible with Windows 3.0

Article ID: Q85178

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Windows version 3.1 introduces the ES_READONLY edit control style. When an application creates a control and specifies the ES_READONLY style, the control retains all the text display properties of an edit control while preserving the content of the control from any modifications.

RROEDIT is a file in the Software/Data Library that demonstrates how to create a read-only edit control using techniques that are compatible with Windows versions 3.0 and 3.1. Using the techniques demonstrated by RROEDIT, an application can switch an edit control to read-only mode and back to normal behavior.

RROEDIT can be found in the Software/Data Library by searching on the word RROEDIT, the Q number of this article, or S13449. RROEDIT was archived using the PKware file-compression utility.

More Information:

The RROEDIT sample subclasses the edit control. In read-only mode, the subclass procedure traps the WM_CHAR, WM_CUT, and WM_PASTE messages and throws them away. All other messages are processed without any intervention from the subclass procedure. The user can copy text from the edit control at all times.

Additional reference words: 3.00 3.10 softlib RROEDIT.ZIP
KBCategory:
KBSubcategory: UsrCtlEdit

PRB: LB_GETCURSEL Function Documentation Incorrect

Article ID: Q70005

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary

SYMPTOMS
   The documentation for the LB_GETCURSEL function on page 6-36 of the
   "Microsoft Windows Software Development Kit Reference Volume 1" for
   versions 3.0, and on page 68 of the "Microsoft Windows Software
   Development Kit Programmer's Reference, Volume 3: Messages,
   Structures, and Macros" for version 3.1 is incorrect.

RESOLUTION
   Listed below is the corrected information for the "Return Value"
   section of the document:

```
   Return Value   The return value is the index of the currently
                  selected item. It is LB_ERR if no item is selected.


   Add the following information as a comment:

   Comment        Do not send this message to a multiple selection
                  list box.
```

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrCtlListbox

FIX: DlgDirSelectComboBox() Fails with Two Combo Box Styles

Article ID: Q70890

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9103005

SYMPTOMS

When the DlgDirSelectComboBox function is used with a CBS_DROPDOWN or CBS_DROPDOWNLIST style combo box, the function will cause the debugging version of Windows to report a fatal exit (RIP). Under enhanced mode, fatal exit 0x0280 is generated; under real and standard modes, fatal exit 0x0007 is generated.

CAUSE

The code did not anticipate the hidden list box that is used in these two styles of controls.

WORKAROUND

To work around this problem, simulate the DlgDirSelectComboBox function by sending the CB_GETCURSEL and CB_GETLBTEXT messages to the combo box.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

More Information:

When a combo box processes a CB_GETCURSEL message, it returns the index of the currently selected item. When a combo box receives a CB_GETLBTEXT message with the currently selected item as a parameter, it returns the text of the item. To completely simulate the function, the application must parse the text returned from the combo box to remove any enclosing square brackets or hyphens.

Additional reference words: 3.00 combobox
KBCategory:
KBSubcategory: UsrCtlOdcombo

INF: Custom Controls Must Use CS_DBLCLKS with Dialog Editor

Article ID: Q71223

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

For a custom control to function properly with the Dialog Editor distributed with version 3.0 of the Microsoft Windows Software Development Kit (SDK), the custom control window class must include the CS_DBLCLKS style.

If the custom control does not have the CS_DBLCLKS style, double-clicking the control in the Dialog Editor does not cause the custom control function to display its style dialog box. However, the control's style dialog box is still accessible from the Styles command on the Edit menu.

More Information:

The Dialog Editor subclasses each control it creates and processes WM_LBUTTONDBLCLK messages. In response to this message, the custom control is asked to display its style dialog box.

If the custom control window class does not have the CS_DBLCLKS style, Windows does not send any WM_LBUTTONDBLCLK messages to the control. As a result, the Dialog Editor does not call the style dialog box function for the custom control and no dialog box appears.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCustomctl

BUG: Buttons Painted Incorrectly After Color Changed
Article ID: Q85593

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows
   version 3.1

SYMPTOMS
========

When the Colors application in the Control Panel is used to change the button shadow color to white, all push button windows are painted incorrectly. Specifically, the Button Shadow and Button Highlight colors are not painted at all.

STATUS
======

Microsoft has confirmed this to be a problem in Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrCtlButtons

INF: Simulating a Sizeable List Box in Windows
Article ID: Q29996

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In Windows 3.0, a list box that has the WS_THICKFRAME style may be sized using the mouse. Although a sizeable list box is not supported in Windows 2.x, it is possible to simulate one by creating a list box in a child window that has a thick frame. The file LISTSIZE in the Software/Data Library contains an example of using this technique.

LISTSIZE can be found in the Software/Data Library by searching on the word LISTSIZE, the Q number of this article, or S12039. LISTSIZE was archived using the PKware file-compression utility.

More Information:

When the child window is sized (that is, the window receives a WM_SIZE message), the list box can be moved to fit exactly within the child window. Under Windows versions 2.x, list boxes can be sized only to an integral number of system-font text heights. Otherwise, the child window would be larger than the list box. In Windows version 3.00, the control style LBS_NOINTEGRALHEIGHT has been defined. Using this style allows the list box to be sized arbitrarily, without regard for the system font height. For backward compatibility, this control style is not used.

In the LISTSIZE example, the maximum vertical height of the child window (and of the list box) is limited by the number of items in the list box. Constraints on the height of the child window also prevent the list box from being sized shorter than two list-box item heights. These limitations prevent any problems resulting from an attempt to size a window smaller than the minimum allowed by Windows.

SetWindowPos() is used to size the child window. MoveWindow() is used to move the list box within the child window. It is possible to change the size of the window borders with the control panel while the application is running. To get the current window-frame height and caption height, GetTextMetrics() is called every time a WM_SIZE message is received.

Window dimensions are necessary because while the client area must be an integral number of system-font text heights, the height passed to SetWindowPos() is the entire window size, including the borders.

Additional reference words: 2.00 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

FIX: EN_CHANGE Not Generated After Edit Control Undo

Article ID: Q71756

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9104003

SYMPTOMS

When an application calls the GetWindowText function after the user enters ALT+BACKSPACE to undo a change to an edit control, incorrect text is copied into the application's buffer.

CAUSE

During the processing of an undo, the EN_CHANGE message is sent to the edit control's parent window at an incorrect time. Undo is performed in two steps:

1. Remove any new text, if necessary.

2. Insert any deleted text, if necessary.

EN_CHANGE is sent after step 1. It should be sent after step 2.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Determining Selected Items in a Multiselection List Box

Article ID: Q71759

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To obtain the indexes of all the selected items in a multiselection list box, the LB_GETSELITEMS message should be sent to the list box.

The message LB_GETCURSEL cannot be used for this purpose because it is designed for use in single-selection list boxes.

Another approach is to send one LB_GETSEL message for every item of the multiselection list box to get its selection state. If the item is selected, LB_GETSEL returns a positive number. The indexes can be built into an array of selected items.

For more information about the LB_GETSELITEMS message, see page 6-38 in the "Windows Software Development Kit Reference Volume 1."

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Data Input Verification for Edit Controls

Article ID: Q72023

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There is a file in the Software/Data Library called VERIFY that contains sample code demonstrating the verification of the contents of a dialog box edit control. The code reads and verifies the edit control string as each character is entered by the user.

More Information:

The contents of the edit control are verified to be legal for a specified data type. If an illegal character is encountered, the character is removed from the edit control and a message beep is generated. The code will verify the input of WORD, DWORD, or FLOAT values.

VERIFY can be found in the Software/Data Library by searching on the word VERIFY, the Q number of this article, or S13072. VERIFY was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: ODVHLB Demonstrates Owner-Draw Variable-Height List Box
Article ID: Q86333

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

ODVHLB is a file in the Software/Data Library that demonstrates using an owner-draw variable-height list box in an application for the Microsoft Windows environment.

In the ODVHLB application, the user can add a short string (one line of text), long string (two lines of text), or an icon to a list box. Because the list box has the LBS_MULTIPLESEL style, the user can select multiple items in the list box simultaneously.

ODVHLB can be found in the Software/Data Library by searching on the word ODVHLB, the Q number of this article, or S13541. ODVHLB was archived using the PKware file-compression utility.

More Information:

To implement an owner-draw variable-height list box, an application must process the WM_MEASUREITEM and WM_DRAWITEM messages in the window procedure for the list box's parent window. When the list box has the LBS_OWNERDRAWVARIABLE style, the parent window receives a WM_MEASUREITEM and a WM_DRAWITEM message each time the application updates an item in the list box.

When the parent of the list box receives a WM_MEASUREITEM message, lParam points to a MEASUREITEMSTRUCT data structure. The application must specify values for the itemHeight and itemWidth members of this structure to indicate how much space in the list box Windows must allocate for the item. Through the WM_MEASUREITEM message, the application can individually specify the height of each item in the list box.

The parent of the list box receives a WM_DRAWITEM message when an item in the list box needs to be drawn. The lParam parameter points to a DRAWITEMSTRUCT data structure. The itemAction member indicates the required drawing action and itemState member indicates the visual state of the item. For example, when the itemAction member is ODA_SELECT and the itemState member has the ODS_SELECTED flag set, the selection state has changed and the item is gaining selection. Conversely, if the itemState member has the ODS_SELECTED flag clear, the item is losing selection.

The itemData member of the DRAWITEMSTRUCT data structure identifies the item in the list box, either through a pointer to a string or a 32-bit identifier. An application specifies an appropriate identifier when it uses the LB_ADDSTRING or LB_INSERTSTRING message to add an

item to the list box.

Additional reference words: 3.10 softlib ODVHLB.ZIP listbox
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Associating Data with a List Box Entry

Article ID: Q74345

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the Microsoft Windows graphical environment, an application can use the LB_SETITEMDATA and LB_GETITEMDATA messages to associate additional information with each entry in a list box. These messages enable an application to associate an arbitrary LONG value with each entry and to retrieve that value. This article documents how an application uses these messages.

More Information:

In this example, the application will associate a 64-byte block of data with each list box entry. This is accomplished by allocating a global memory block and using the LB_SETITEMDATA message to associate the handle of the memory block with the appropriate list box item.

During list box initialization, the following code is executed for each list box item:

```
   if ((hLBData = GlobalAlloc(GMEM_MOVEABLE, 64)))
      {
      if ((lpLBData = GlobalLock(hLBData)))
         {
         // Store data in 64-byte block.

         GlobalUnlock(hLBData);
         }
      }
   SendMessage(hListBox, LB_SETITEMDATA, nIndex,
         MAKELONG(hLBData, 0));
```

To retrieve the information associated with a list box entry, the following code can be used:

```
   if ((hLBData = LOWORD(SendMessage(hListBox, LB_GETITEMDATA,
           nIndex, 0L))))
      {
      if ((lpLBData = GlobalLock(hLBData)))
         {
         // Access or manipulate the data or both.

         GlobalUnlock(hLBData);
         }
      }
```

Before the application terminates, it must free the memory associated

with each list box item. The following code frees the memory associated with one list box item:

```
   if ((hLBData = LOWORD(SendMessage(hListBox, LB_GETITEMDATA,
           nIndex, 0L))))
      GlobalFree(hLBData);
```

These techniques can be used to associated data with an entry in a combo box by substituting the CB_SETITEMDATA and CB_GETITEMDATA messages.

Additional reference words: 3.00 combobox listbox
KBCategory:
KBSubcategory: UsrCtlListbox

BUG: Combo Boxes in DS_SYSMODAL Dialog Boxes

Article ID: Q74507

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9107004

SYMPTOMS
   When a combo box with the CBS_DROPDOWN or CBS_DROPDOWNLIST style is
   used inside a system modal dialog box, it causes the following
   three problems:

   1. An item cannot be selected from the drop-down list box by
      clicking it with the mouse. The original selection remains in
      the edit control or static-text control.

   2. The drop-down list box cannot be closed by clicking on a part of
      the list box that lies outside the dialog box.

   3. The parts of the drop-down list box that lie outside the dialog
      box are not erased when the drop-down list box is closed.

CAUSE
   The mouse messages meant for the drop-down list box never reach the
   list box.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. We are researching this problem and will post new information
   here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCombo

FIX: Drawing Problem When Group Box Text Changes
Article ID: Q74510

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9107005

SYMPTOMS
Under version 3.0 of the Microsoft Windows graphical environment, when an application calls the SetDlgItemText function to change the text of a group box, any text not overwritten by the new text continues to be displayed.

CAUSE
A group box is implemented as a type of transparent window with a border and title text. A group box cannot erase its background because doing so would erase any controls inside the box. Versions of Windows earlier than 3.1 do not erase the portion of the group box under the title text.

RESOLUTION
Microsoft has confirmed this to be a problem in Windows version 3.0. Two methods to work around this problem are as follows:

- Be sure that the string passed to SetDlgItemText is as long as or longer than the text displayed in the control. If necessary, add blanks to the end of the string.

  -or-

- Cause the application to repaint the group box text. The following code fragment implements this method:

```
char temp_string[MAXLEN];
RECT rect;
HWND hDlg, hGroupBox;

SetDlgItemText(hDlg, ID_GROUPBOX, temp_string);
hGroupBox = GetDlgItem(hDlg, ID_GROUPBOX);
GetWindowRect(hGroupBox, &rect);
ScreenToClient(hDlg, (LPPOINT)&rect.left);
ScreenToClient(hDlg, (LPPOINT)&rect.right);
InvalidateRect(hDlg, &rect, TRUE); // erase background
UpdateWindow(hDlg); // repaint
```

This problem was corrected in Windows version 3.1. The methods provided above to avoid the problem are compatible with Windows 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlStatic

INF: Graying the Text of a Button or Static Text Control

Article ID: Q39480

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

A control is a child window that is responsible for processing keyboard, mouse, focus, and activation messages, among others. A control paints itself and processes text strings. The color of the text in a button or static text control is automatically changed to gray when the control is disabled with the EnableWindow function. However, If an application subclasses a control to process WM_PAINT messages for the control, the application can use the GrayString function to change the text color.

Additional reference words: 2.10 3.00 3.10 grey SR# G881208-7678
KBCategory:
KBSubcategory: UsrCtlStatic

FIX: EM_SETWORDBREAK Not Implemented in Windows 3.0
Article ID: Q74600

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9107008

SYMPTOMS

The EM_SETWORDBREAK message is documented in the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0; however, this message is not implemented in Windows 3.0.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. Windows 3.1 implements the EM_SETWORDBREAKPROC message, which provides a more efficient interface through which Windows can provide word break information.

Additional reference words: 3.00 SDK
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Limit to the Number of Characters Stored in a List Box

Article ID: Q39544

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The memory limit for a Windows list box depends strictly on the number of characters, not on the number of items in the list box. The memory for the strings is allocated from global memory, but the internal code assumes that the total amount of memory required by the strings in the list box will not exceed 32K (in Windows versions 2.x). If the total memory required by the strings is more than 32K, the list box will not function correctly.

Under Windows version 3.0, the limit is 64K of text.

Additional reference words: 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Determining the Visible Area of a Multiline Edit Control

Article ID: Q88387

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary :

The multiline edit control provided with Microsoft Windows versions 3.0 and 3.1 does not provide a mechanism that allows an application to determine the currently visible lines of text. This article outlines an algorithm to provide that functionality.

More Information:

The general idea is to determine the first and last visible lines and obtain the text of those lines from the edit control. The following steps detail this process:

1. A Windows-3.1-based application can use the newly available message, EM_GETFIRSTVISIBLELINE, to determine the topmost visible line.

2. Obtain the edit control's formatting rectangle using EM_GETRECT. Determine the rectangle's height using this formula:

   ```
   nFmtRectHeight = rect.bottom - rect.top;
   ```

3. Obtain the line spacing of the font used by the edit control to display the text. Use the WM_GETFONT message to determine the font used by the edit control. Select this font into a display context and use the GetTextMetrics function. The tmHeight field of the resulting TEXTMETRIC structure is the line spacing.

4. Divide the formatting rectangle's height (step 2) with the line spacing (step 3) to determine the number of lines. Compute the line number of the last visible line based on the first visible line (step 1) and the number of visible lines.

5. Use EM_GETLINE for each line number from the first visible line to the last visible line to determine the visible lines of text. Remember that the last visible line may not necessarily be at the bottom of the edit control (the control may only be half full). To detect this case, use EM_GETLINECOUNT to know the last line and compare its number with the last visible line. If the last line number is less than the last visible line, your application should use EM_GETLINE only on lines between the first and the last line.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Making a List Box Item Unavailable for Selection
Article ID: Q74792

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

In the Microsoft Windows graphical environment, an application can use a list box to enumerate options. However, there are circumstances in which one or more options may not be appropriate. The application can change the appearance of items in a list box and prevent the user from selecting one of these items by using the techniques discussed below.

More Information:

Changing the Appearance of a List Box Item

To dim (gray) a particular item in a list box, use an owner-draw list box as follows:

1. Create a list box that has the LBS_OWNERDRAW and LBS_HASSTRINGS styles.

2. Use the following code to process the WM_MEASUREITEM message:

```
case WM_MEASUREITEM:
   ((MEASUREITEMSTRUCT FAR *)(lParam))->itemHeight = wItemHeight;
   break;
```

   wItemHeight is the height of a character in the list box font.

3. Use the following code to process the WM_DRAWITEM message:

```
 #define PHDC (pDIS->hDC)
 #define PRC  (pDIS->rcItem)

 DRAWITEMSTRUCT FAR *pDIS;

   ...

 case WM_DRAWITEM:
   pDIS = (DRAWITEMSTRUCT FAR *)lParam;

   SendMessage(pDIS->hwndItem, LB_GETTEXT, pDIS->itemID,
               (LONG)(LPSTR)szBuff);

   FillRect(PHDC, &PRC,
               GetClassWord(pDIS->hwndItem, GCW_HBRBACKGROUND));

   if ('!' == *szBuf) // this string is disabled
```

```
        GrayString(PHDC, hGrayBrush, NULL, szBuf + 1, 0, 0, 0, 0, 0);
      else
        TextOut(PHDC, PRC.left, PRC.top, szBuf, lstrlen(szBuf));

      if ((pDIS->itemState) & (ODS_FOCUS))
        DrawFocusRect(PHDC, &PRC);

      return TRUE;
```

Strings that start with "!" are displayed dimmed. The exclamation mark character is not displayed.

Preventing Selection

To prevent a dimmed string from being selected, create the list box with the LBS_NOTIFY style. Then use the following code in the list box's parent window procedure to process the LBN_SELCHANGE notification:

```
  case WM_COMMAND:

    switch (wParam)
      {

       ...

      case IDD_LISTBOX:
        if (LBN_SELCHANGE == HIWORD(lParam))
          {
          idx = (int)SendDlgItemMessage(hDlg, wParam,
                        LB_GETCURSEL, 0, 0L);
          SendDlgItemMessage(hDlg, wParam, LB_GETTEXT, idx,
                        (LONG)(LPSTR)szBuf);
          if ('!' == *szBuf)
            {
            // Calculate an alternate index here
            // (not shown in this example).

            // Then set the index.
            SendDlgItemMessage(hDlg, wParam, LB_SETCURSEL, idx, 0L);
            }
          }
        break;

       ...

      }
   break;
```

When the user attempts to select a dimmed item, the alternate index calculation moves the selection to an available item.

Additional reference words: 3.00 3.10 listbox
KBCategory:
KBSubcategory: UsrCtlOdlistbox

INF: Buttons and Cursors Documentation and Sample
Article ID: Q89562

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

BTTNCUR.ZIP is a file in the Software/Data Library that contains information about using buttons and cursors in the Microsoft Windows graphical environment. BTTNCUR.ZIP contains the source code to a dynamic-link library (DLL) that contains 15 new standard cursors and 9 standard command images to use as buttons on an application's toolbar. Other code in the DLL draws a button using one of these images (or another image provided by an application) in one of six different states. With this code, an application can provide only one image for a button and dynamically create other states at run time. This dramatically reduces the storage required for an application to provide a button-rich graphical interface.

BTTNCUR.ZIP can be found in the Software/Data Library by searching on the keyword BTTNCUR, the Q number of this article, or S13594. BTTNCUR.ZIP was archived using the PKware file-compression utility.

The BTTNCUR.ZIP file contains a directory structure. To reproduce the directory structure, specify the -d option to the PKUNZIP command, as follows:

```
   pkunzip -d bttncur
```

Additional reference words: 3.10 softlib BTTNCUR.ZIP
KBCategory:
KBSubcategory: UsrCtlButtons

INF: Sample Code to Demonstrate a Button Bar
Article ID: Q74999

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The file BTNBAR.ZIP in the Software Library demonstrates one method of adding a button bar to an application, such as the ToolBar, which is found in Microsoft Excel. ToolBar is a registered trademark of Microsoft Corporation.

More Information:

For the button bar to correctly interact with the MDI (multiple document interface) functions of Windows, the client window must be resized to avoid painting problems. The code in the BTNBAR file is designed to be merged into the MULTIPAD sample application that is provided with the Windows Software Development Kit (SDK) version 3.0.

The remainder of this article provides some information on the design of this button bar implementation.

Each button on the button bar is defined in an array of TOOL structures, which is defined as follows:

```
  typedef struct tagTOOL
    {
    HICON     hIcon;             // 2
    WORD      CommandID;         // 2
    BOOL      bEnabled;          // 2
    WORD      x, y, dx, dy;      // 8
    }                            // --
  TOOL;                          // 14 bytes total
```

To conserve system resources, the determination that a given button has been pressed is done by performing hit testing, instead of creating separate buttons. Hit testing is demonstrated in Charles Petzold's "Checkers" program, presented in the "Microsoft Systems Journal."

This implementation of the button bar uses icons to label the buttons. The DrawTool() function draws a plain button with a gray face and dark edges, to simulate a three-dimensional object. Then, the icon specified in the structure is drawn over the button. This icon must use the "screen" color for its background color. This places the icon image on the button with minimal coding effort.

When the user clicks on a button, the button bar code sends a WM_COMMAND message to the main window (ghWnd). The wParam parameter of this message is set to the CommandID value.

BTNBAR can be found in the Software/Data Library by searching on the keyword BTNBAR, the Q number of this article, or S13129. BTNBAR was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlCustomctl

INF: Altering Edit Control Strings in Place May Cause UAE

Article ID: Q75500

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Declaring an Edit control with certain styles, and altering the text, may have undesirable effects. Depending upon whether or not pointers to the text are passed to other functions, changing the text in place may generate an unrecoverable application error (UAE).

Declaring an Edit control with any of the following styles, ES_LOWERCASE, ES_UPPERCASE, or ES_OEMCONVERT, and calling either SetWindowText (hEdt, lpstr) or SetDlgItemText (hDlg, editID, lpstr) will alter the string in place. Calling either one of the above mentioned functions to alter a string contained in a code segment, which is labeled PURE in the corresponding .DEF file, will result in a UAE.

Additional reference words: 3.00 controls strings
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Changing an Edit Control to a Bedit Control at Run Time

Article ID: Q90839

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

PENWIN.DLL by default changes an edit control to an hedit control at run time. However, the Pen Windows system does not have a built-in mechanism to change an edit control to a bedit control at run time.

More Information:

DYNBEDIT.ZIP demonstrates how to change a normal edit control to a bedit control at run time. This allows an application to run unmodified in either Pen Windows or retail Windows.

DYNBEDIT is located in the Software/Data Library and can be found by searching on the word DYNBEDIT, the Q number of this article, or S13705. DYNBEDIT was archived using the PKware file-compression utility.

The basic steps to perform this transition are to query the information from the edit control, destroy the edit control, and then call CreateWindow() to make the new bedit control. However, there is a problem with this procedure when using a font other than the system default font in a dialog box. If a font statement is given (for example: FONT 8,"Helv"), the combs of the bedit control will appear below the bottom border of the control. This problem occurs because the Pen system bases the size of the combs on the system font only rather than on the font specified. To work around this problem, the application must modify the GUIDE structure in the RC structure for the new control. By reducing the size of the GUIDE structure, the Pen system will pick a new font, which matches the new GUIDE size.

To accomplish this, the following steps need to be performed at either WM_INITDIALOG or WM_CREATE depending upon the location of the edit control:

1. Check to see if Pen Windows is running. If so, proceed to step 2.

2. Get the handle to the edit control, the text in the edit control, and three different sets of dimensions:

   a. The location of the dialog box in screen coordinates.
   b. The location of the edit control in screen coordinates.
   c. The size of the edit control in pixels.

   The first two measurements can be accomplished with the GetWindowRect function, and the third measurement can be accomplished with the GetClientRect function.

3. Destroy the original edit control.

4. Create the new bedit control with a call to CreateWindow(). Use the following equations to calculate the values of x, y, dx, and dy:

```
// EditRect is the location of the edit control in screen
// coordinates.
// DlgRect is the location of the dialog box in screen coordinates.
// rect is the size of the edit control in pixels.

x  = (EditRect.left - DlgRect.left)
y  = (EditRect.top - DlgRect.top)
dx = (rect.right - rect.left)
dy = (rect.bottom - rect.top)
```

   You should also use the same ID number in the call to CreateWindow() as used by the original edit control.

5. Send a WM_HEDITCTL message to the new bedit control with the wParam set to HE_GETRC to retrieve the RC structure. This will allow the application to modify the GUIDE structure.

6. Calculate the new cyBox value. cyBox is the height of one cell in the bedit control. For example:

```
// New cyBox value (height of one box).
NewcyBox = (rect.bottom - rect.top);
```

7. Calculate the new cxBox value by using a ratio between the old cyBox and the new cyBox. cxBox is the width of one cell in the control. For example:

```
// Calculate new cxBox value by changing window ratio.
rc.guide.cxBox = NewcyBox * rc.guide.cxBox / rc.guide.cyBox;
```

8. Calculate the new cyBase value. cyBase is the distance from the top of the cell to the baseline of the comb. Again, use a ratio based upon the change between the old cyBox and the new cyBox. For example:

```
// Calculate the new cyBase (distance from top of box to baseline).
rc.guide.cyBase = NewcyBox * rc.guide.cyBase / rc.guide.cyBox;
```

9. Calculate the number of cells that can now fit inside of the control. This is done by taking the width of the control, and dividing it by the width of a single cell. For example:

```
// Calculate the number of cells that can fit in the bedit control.

rc.guide.cHorzBox = (rect.right - rect.left) / rc.guide.cxBox;
```

10. Complete any additional changes needed for the control, and then replace the RC structure with the new one by sending a WM_HEDITCTL message with the wParam set to HE_SETRC. For example:

```
// Replace the RC structure.
SendMessage (hBEdit, WM_HEDITCTL, HE_SETRC, (long)(LPRC) &rc);
```

At this point, the Pen system will generate the correct font to fit inside of the new GUIDE dimensions.

If the application chooses to send a WM_SETFONT message to the control, the application then assumes responsibility for choosing a font that will fit inside of the comb. The Pen system won't do any font picking if the application uses the WM_SETFONT message. Using a WM_SETFONT message is necessary if the application requires the use of any special font characteristics (for example, italic).

The DYNBEDIT.ZIP sample has been written to demonstrate the above process. The code that makes the transition is located in ABOUT.C in the WM_INITDIALOG case of PenDlgProc().

Additional reference words: 3.10 DYNBEDIT PEN runtime
KBCategory:
KBSubcategory: UsrCtlEdit

PRB: Accented Characters in Filename Controls Lose Accents

Article ID: Q90854

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.0
   and 3.1

Summary:

SYMPTOMS

In an application for the Microsoft Windows graphical environment that uses a combo box with the CBS_OEMCONVERT style or an edit control with the ES_OEMCONVERT style, the user enters a character with an accent mark, and the accent is lost.

CAUSE

The OEMCONVERT control styles convert all input as follows:

- Convert lowercase letters to uppercase letters with the AnsiUpper function.

- Convert all characters from the ANSI character set to the installed OEM character set with the AnsiToOem function.

- Convert the appropriate characters back to lowercase letters with the AnsiLower function.

Because the OEM character set installed in most computers (code page 437) does not include representation for the capital letters with accent marks, the accent marks are lost in this conversion process. This occurs in all versions of Windows, including those designed for use outside the United States.

RESOLUTION

Unicode specifies a unified character set that can represent every active language and many dead languages. Windows NT incorporates support for Unicode.

More Information:

For further details on the cause of this problem, see the article by Dr. William S. Hall "Adapt Your Program for Worldwide Use with Windows Internationalization Support," starting on page 29 of the Nov-Dec 1991 Microsoft Systems Journal.

Additional reference words: 2.00 2.03 2.10 2.x 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlCombo

PRB: WM_SETTEXT Ignores EM_LIMITTEXT Edit Control Text Limit

Article ID: Q75865

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

When the WM_SETTEXT message is used to place text into an edit control, more text than the text length limit specified in an EM_LIMITTEXT message can be placed into the control.

RESOLUTION

This is the designed behavior for these messages. This allows an application to create an edit control where only the first "n" characters can be entered by the user, and the remainder of the text is specified by the application ("n" is specified in the EM_LIMITTEXT message).

More Information:

There is a limitation to this method. If the application specifies more text than is specified in the EM_LIMITTEXT message, the user can edit the entire contents of the edit control.

There are two ways to work around this limitation:

1. Erase all text from the edit control before allowing the user to enter text.

2. After each WM_SETTEXT message, truncate the edit control text to less than the limit specified in the EM_LIMITTEXT message.

The edit control will enforce the length limit if the amount of text in the control is less than the limit. Using a WM_SETTEXT message allows the application to violate the limit.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Creating the Default Border Around a Push Button

Article ID: Q48713

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There is a file named DEFBTN in the Software/Data Library that demonstrates the correct message sequence to change the status of a push button from the "normal" state to the "default" state. The default push button receives a message when the ENTER key is pressed in a dialog box, no matter which control currently has the focus.

In the DEFBTN example, when a numeral button is pressed ("1", "2", or "3") the corresponding number button ("one", "two", or "three") becomes the default button.

DEFBTN can be found in the Software/Data Library by searching on the keyword DEFBTN, the Q number of this article, or S12379. DEFBTN was archived using the PKware file-compression utility.

For a button to be correctly activated by using an accelerator, it is necessary to subclass the button to process the focus messages. This is caused by the fact that accelerators bypass some steps in the message flow.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlButtons

PRB: Moving or Resizing the Parent of an Open Combo Box
Article ID: Q76365

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

When the user resizes or moves the parent window of an open drop-down combo box, the list box portion of the combo box does not move.

CAUSE

The list box portion of the combo box does not receive a move message. Therefore, it remains on the screen at its original position.

RESOLUTION

Close the drop down list before the combo box is moved. To perform this task, during the processing of the WM_PAINT message, send the combo box a CB_SHOWDROPDOWN message with the wParam set to FALSE.

Additional reference words: 3.00 3.10 3.x combobox
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Switching Between Single and Multiple List Boxes
Article ID: Q57959

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

After creating a list box with the CreateWindow function, changing the list box from a single selection to a multiple selection can be accomplished in the following way:

Create two hidden list boxes in the .RC file, and during the WM_INITDIALOG routine, display one of the boxes. Change between the two by making one hidden and the other one visible using the ShowWindow function.

Additional reference words: 2.03 2.10 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlListbox

PRB: Button Styles May Not Be Combined
Article ID: Q76933

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Resource script statements are documented in Chapter 8 of the "Microsoft Windows Software Development Kit Reference Volume 2." This chapter includes a discussion of button controls and the following statement on page 8-24:

> In addition to these styles, the style field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

This statement, which is repeated a number of times throughout the chapter, is incorrect; button styles cannot be combined.

More Information:

The button style identifiers (BS_*) are defined in the WINDOWS.H file as consecutive integers, not individual bits. If two button styles are combined and the combination creates a legal button style, the button will be created with that style. However, if two button styles are combined and the combination does not create a legal button style, the button will be created, but will not be painted on the screen.

Microsoft has confirmed this to be an error in version 3.0 of the Windows Software Development Kit (SDK) documentation.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlButtons

INF: Extending Standard Windows Controls Through Superclassing

Article ID: Q76947

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A Windows application can extend the behavior of a standard Windows control by using the technique of superclassing. An application can superclass a standard Windows control by retrieving its window class information, modifying the fields of the WNDCLASS structure, and registering a new class. For example, to associate status information with each button control in an application, the buttons can be superclassed to provide a number of window extra bytes.

This article describes a technique to access the WNDCLASS structure associated with the standard "button" class.

More Information:

The following five steps are necessary to register a new class that uses some information from the standard windows "button" class:

1. Call GetClassInfo() to fill the WNDCLASS structure.

2. Save the cbWndExtra value in a global variable.

3. Add the desired number of bytes to the existing cbWndExtra value.

4. Change the lpszClassName field.

5. Call RegisterClass() to register the new class.

The first step will fill the WNDCLASS structure with the data that was used when the class was originally registered. In this example, the second step is necessary so that when the "new" extra bytes are accessed, the original extra bytes are not destroyed. Please note that it is NOT safe to assume that the original cbWndExtra value was zero. When accessing the "new" extra bytes, it is necessary to use the original value of cbWndExtra as the base for any new data stored in the extra bytes. The third step allocates the new extra bytes. The fourth step specifies the new name of the class to be registered, and the final step actually registers the new class.

Any new class created in this manner MUST have a unique class name. Typically, this name would be similar but not identical to the original class. For example, to superclass a button, an appropriate class name might be "superbutton." There is no conflict with class names used by other applications as long as the CS_GLOBALCLASS class style is not specified. The standard Windows "button" class remains unchanged and can still be used by the application as normal. In

addition, once a new class has been registered, any number of controls can be created and destroyed with no extra coding effort. The superclass is simply another class in the pool of classes that can be used when creating a window.

The sample code below demonstrates this procedure:

```
BOOL DefineSuperButtonClass(void)
{
#define MYEXTRABYTES 8

   HWND     hButton;
   WNDCLASS wc;

   GetClassInfo(NULL, "button", (LPWNDCLASS)&wc);

   iStdButtonWndExtra = wc.cbWndExtra;   // Save this in a global

   wc.cbWndExtra += MYEXTRABYTES;

   lstrcpy((LPSTR)wc.lpszClassName, (LPSTR)"superbutton");

   return(RegisterClass((LPWNDCLASS)&wc));
}
```

It is important to note that the lpszClassName, lpszMenuName, and hInstance fields in the WNDCLASS structure are NOT returned by the GetClassInfo() function. Please refer to page 4-153 of the "Microsoft Windows Software Development Kit Reference Volume 1" for more information. Also, each time a new class is registered, scarce system resources are used. If it is necessary to alter many different standard classes, the GetProp(), SetProp(), and RemoveProp() functions should be used as an alternative approach to associating extra information with standard Windows controls.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlSuperclass

FIX: Resizing Multiline Edit Control Causes UAE
Article ID: Q77261

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9110008

SYMPTOMS
   When a multiline edit control contains more than one line of text
   and is resized so that the text wraps differently, the application
   experiences an unrecoverable application error (UAE).

CAUSE
   When the edit control is resized, not all internal data structures
   are updated to reflect the new size.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEdit

INF: Raising Text Size Limit for Edit Controls
Article ID: Q77287

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

By default, the buffer for an edit controls is allocated from the application's data segment. Because the data segment is also used to store information for other controls, space in the data segment is a scarce resource.

GLBEDIT is a code sample in the Software/Data Library that demonstrates how an edit control can have its own data segment. GLBEDIT can be found in the Software/Data Library by searching on the keyword GLBEDIT, the Q number of this article, or S13200. GLBEDIT was archived using the PKware file-compression utility.

More Information:

Normally, edit controls in a dialog box use one common global data segment per dialog box. If the dialog box has style DS_LOCALEDIT, edit controls use the data segment pointed to by the hInstance used to create the dialog box. Edit controls created with CreateWindow() use the data segment pointed to by hInstance.

The GLBEDIT sample shows how to change hInstance to point to a new global object. Multiple controls of any type can use the same memory object.

To find the effected portions of the code, search for WM_CREATE in the MainWndProc.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlEditbuff

INF: Subclassing Sample Code Available in Software Library
Article ID: Q77480

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The file TRACK.ZIP, which contains a program to demonstrate subclassing controls in Windows, is available in the Software Library. TRACK.ZIP can be found in the Software/Data Library by searching on the word TRACK, the Q number of this article, or S12038. TRACK was archived using the PKware file-compression utility.

More Information:

Subclassing is a powerful technique that an application can use to modify the messages sent to a control. Subclassing can be used in a variety of ways. It can be used to change the color of the text in a list box or to create an edit control that echoes only asterisks and can be used for accepting passwords.

Subclassing must be used with care. Because the order in which messages are sent to standard controls is subject to change as Windows is revised, an application tied to message order may fail in a future version of Windows. Subclassing should not be used to alter the behavior of a standard control.

For more information on subclassing in Windows, query on the following words:

    prod(winsdk) and subclassing

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrCtlSubclass

INF: DlgDirList on Novell Drive Doesn't Show Double Dots [..]

Article ID: Q99339

Summary:

The Microsoft Windows application programming interface (API) function DlgDirList() can be used to add directories, drives, and/or files to a list box. On a Novell network with the default login script, the Windows API may not add the string "[..]" used to represent the parent directory to the list box. This is due to Novell implementation, and is not a bug in the Windows API. To make the entry appear, add the following line to the Novell login script (usually called SHELL.CFG located in the root directory of the boot drive)

```
   SHOW DOTS = ON
```


More Information:

A Novell NetWare file server does not include the directory entries dot (.) and double dot (..) as MS-DOS does. However, the NetWare shell (version 3.01 or later) can emulate these entries when applications attempt to list the files in a directory. Turning on Show Dots causes problems with earlier versions of some 286-based NetWare utilities, such as BINDFIX.EXE and MAKEUSER.EXE. Make sure you upgrade these utilities if you upgrade your NetWare shell. For more information, contact your Novell dealer.

Note: With Novell NetWare version 3.1.1, the line SHOW DOTS=ON/OFF can be added to the NET.CFG file for the same effect.

The same behavior is shown with the API DlgDirListComboBox, and the messages LB_DIR and CB_DIR.

Additional reference words: 3.10 up missing UsrCtlCombo listbox
KBCategory:
KBSubcategory: UsrCtlListbox

INF: Safe Subclassing
Article ID: Q101180

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1

SUMMARY
=======

This article describes subclassing, how it is done, and the rules that should be followed to make subclassing safe. Both instance and global subclassing are covered. Superclassing is described as an alternative to global subclassing.

MORE INFORMATION
================

Subclassing Defined

Subclassing is a technique that allows an application to intercept messages destined for another window. An application can augment, monitor, or modify the default behavior of a window by intercepting messages meant for another window. Subclassing is an effective way to change or extend the behavior of a window without redeveloping the window. Subclassing the default control window classes (button, edit, list box, combo box, static, and scroll bar controls) is a convenient way to obtain the functionality of the control and to modify its behavior. For example, if a multiline edit control is included in a dialog box and the user presses the ENTER key, the dialog box closes. By subclassing the edit control, an application can have the edit control insert a carriage return and linefeed into the text without exiting the dialog box. An edit control does not have to be developed specifically for the needs of the application.

The Basics

The first step in creating a window is registering a window class by filling a WNDCLASS structure and calling RegisterClass. One element of the WNDCLASS structure is the address of the window procedure for this window class. When a window is created, the Microsoft Windows graphical environment takes the address of the window procedure in the WNDCLASS structure and copies it to the new window's information structure. When a message is sent to the window, Windows calls the window procedure through the address in the window's information structure. To subclass a window, you substitute a new window procedure that receives all the messages meant for the original window by substituting the window procedure address with the new window procedure address.

When an application subclasses a window, it can take three actions

with the message: (1) pass the message to the original window procedure; (2) modify the message and pass it to the original window procedure; (3) not pass the message. The application subclassing a window can decide when to react to the messages it receives. The application can process the message before, after, or both before and after passing the message to the original window procedure.

Types of Subclassing

The two types of subclassing are instance subclassing and global subclassing. Instance subclassing is subclassing an individual window's information structure. With instance subclassing, only the messages of a particular window instance are sent to the new window procedure. Global subclassing is replacing the address of the window procedure in the WNDCLASS structure of a window class. All subsequent windows created with this class have the substituted window procedure's address. Global subclassing affects only windows created after the subclass has occurred. If any windows exist of the window class that is being globally subclassed at the time of the subclass, the existing windows are not affected by the global subclass. If the application needs to affect the behavior of the existing windows, the application must subclass each existing instance of the window class.

Instance Subclassing

The SetWindowLong function is used to subclass an instance of a window. The application must have the procedure-instance address of the subclass function. The subclass function is the function that receives the messages from Windows and passes the messages to the original window procedure. To get the procedure-instance address of the subclass function, the application calls MakeProcInstance. If the subclass function resides in a dynamic-link library (DLL), the application does not need to call MakeProcInstance to get the procedure-instance address. The subclass function must be exported in the application's or the DLL's module definition file.

The application subclassing the window calls SetWindowLong with the handle to the window the application wants to subclass, the GWL_WNDPROC option (defined in WINDOWS.H), and the procedure-instance address of the new subclass function. SetWindowLong returns a DWORD, which is the address of the original window procedure for the window. The application must save this address to pass the intercepted messages to the original window procedure and to remove the subclass from the window. The application passes the messages to the original window procedure by calling CallWindowProc with the address of the original window procedure and the hWnd, Message, wParam, and lParam parameters used in Windows messaging. Usually, the application simply passes the arguments it receives from Windows to CallWindowProc.

The application also needs the original window procedure address for removing the subclass from the window. The application removes the subclass from the window by calling SetWindowLong again. The application passes the address of the original window procedure with the GWL_WNDPROC option and the handle to the window being subclassed. The following code subclasses and removes a subclass to an edit

control:

```
LONG FAR PASCAL SubClassFunc(HWND hWnd,WORD Message,WORD wParam,
      LONG lParam);
FARPROC lpfnOldWndProc;
HWND hEditWnd;

//
// Create an edit control and subclass it.
// The details of this particular edit control are not important.
//
hEditWnd = CreateWindow("EDIT", "EDIT Test",
                        WS_CHILD | WS_VISIBLE | WS_BORDER ,
                        0, 0, 50, 50,
                        hWndMain,
                        NULL,
                        hInst,

                        NULL);
//
// Now subclass the window that was just created.
//
lpfnSubClassProc=MakeProcInstance((FARPROC) SubClassFunc,hInst);
lpfnOldWndProc =
   (FARPROC)SetWindowLong(hEditWnd, GWL_WNDPROC, (DWORD)
      lpfnSubClassProc);
.
.
.
//
// Remove the subclass for the edit control.
//
SetWindowLong(hEditWnd, GWL_WNDPROC, (DWORD) lpfnOldWndProc);

//
// Here is a sample subclass function.
//
LONG FAR PASCAL SubClassFunc(   HWND hWnd,

               WORD Message,
               WORD wParam,
               LONG lParam)
{
   //
   // When the focus is in an edit control inside a dialog box, the
   //  default ENTER key action will not occur.
   //

   if ( Message == WM_GETDLGCODE )
       return DLGC_WANTALLKEYS;

   return CallWindowProc(lpfnOldClassProc, hWnd, Message, wParam,
                         lParam);
}
```

Potential Pitfalls

Instance subclassing is generally safe, but observing the following rules ensures safety. When subclassing a window, you must know what owns the window's behavior. For example, Windows owns all controls it supplies; applications own all windows they define. Subclassing can be done on any window in the system; however, when an application subclasses a window that the application does not own, the application must ensure that the subclass function does not break the original behavior of the window. Because the application does not control the window, it should not rely on any information about the window that the owner might change in the future. A subclass function should not use the extra window bytes or the class bytes for the window unless it knows exactly what they mean and how the original window procedure uses them. Even if the application knows everything about the extra window bytes or the class bytes, it should not use them unless the application owns the window. If an application uses the extra window bytes of a window that another application owns and the owner decides to update the window and change some aspect of the extra bytes, the subclass procedure is likely to fail. For this reason, Microsoft suggests that you not subclass the control classes. Windows owns the controls it supplies, and aspects of the controls might change from one version of Windows to the next. If your application must subclass a control supplied by Windows, it may need to be updated when a new version of Windows is released.

Because instance subclassing occurs after a window is created, the application subclassing the window cannot add any extra bytes to the window. Applications that subclass windows should store any data needed for an instance of the subclassed window in the window's property list. The SetProp function attaches properties to a window. The application calls SetProp with the handle to a window, a string identifying the property, and a handle to the data. The handle to the data is usually obtained with a call to either LocalAlloc or GlobalAlloc. When the application uses the data in a window's property list, the application calls the GetProp function with the handle to the window and the string that identifies the property. GetProp returns the handle to the data that was set with SetProp. When the application is finished with the data or when the window is to be destroyed, the application must remove the property from the property list by calling the RemoveProp function with the handle to the window and the string identifying the property. RemoveProp returns the handle to the data, which the application then uses in a call to either LocalFree or GlobalFree. The Microsoft Windows SDK "Programmer's Reference, Volume 2, Functions" manual describes SetProp, GetProp, and RemoveProp.

When an application subclasses a subclassed window, the subclasses must be removed in reverse of the order in which they were performed.

Global Subclassing

Global subclassing is similar to instance subclassing. The application calls SetClassLong to globally subclass a window class. As it does with instance subclassing, the application needs the procedure-instance address of the subclass function, and the subclass function must be exported in the application's or the DLL's module

definition file. To globally subclass a window class, the application must have a handle to a window of that class. To get a handle to a window in the desired class, most applications create a window of the class to be globally subclassed. When the application removes the subclass, it needs a handle to a window of the type the application wants to subclass, so creating and keeping a window for this purpose is the best technique. If the application creates a window of the type it wants to subclass, the window is usually hidden. After obtaining a handle to a window of the correct type, the application calls SetClassLong with the window handle, the GCL_WNDPROC option (defined in WINDOWS.H), and the procedure-instance address of the new subclass function. SetClassLong returns a DWORD, which is the address of the original window procedure for the class. The original window procedure address is used in global subclassing in the same way it is used in instance subclassing. Messages are passed to the original window procedure in the same way as in instance subclassing, by calling CallWindowProc. The application removes the subclass from the window class by calling SetClassLong again. The application passes the address of the original window procedure with the GCL_WNDPROC option and a handle to the window of the class being subclassed. An application that globally subclasses a control class must remove the subclass when the application finishes.

The following code globally subclasses and removes a subclass to an edit control:

```
LONG FAR PASCAL SubClassFunc(HWND hWnd,WORD Message,WORD wParam,
      LONG lParam);
FARPROC lpfnOldClassWndProc;
HWND hEditWnd;

//
// Create an edit control and subclass it.
// Notice that the edit control is not visible.
// Other details of this particular edit control are not important.
//
hEditWnd = CreateWindow("EDIT", "EDIT Test",
                        WS_CHILD,
                        0, 0, 50, 50,
                        hWndMain,
                        NULL,

                        hInst,
                        NULL);
lpfnSubClassProc=MakeProcInstance((FARPROC) SubClassFunc,hInst);
lpfnOldClassWndProc =
   (FARPROC)SetClassLong(hEditWnd, GCL_WNDPROC, (DWORD)
      lpfnSubClassProc);
.
.
.
//
// To remove the subclass:
//
SetClassLong(hEditWnd, GWL_WNDPROC, (DWORD) lpfnOldClassWndProc);
DestroyWindow(hEditWnd);
```

Potential Pitfalls

Global subclassing has the same limitations as instance subclassing but presents some additional problems. The application should not attempt to use the extra bytes for either the class or the window instance unless it knows exactly how the original window procedure is using them. If data must be associated with a window, the window's properties list should be used in the same way as in instance subclassing. Global subclassing is dangerous when used with the Windows-supplied control classes, especially if more than one application globally subclasses a control class.

The following sequence of events illustrates this problem. Application A globally subclasses the edit control class and stores the original window procedure's address. Application B then also globally subclasses the edit control class. When application B calls SetClassLong with the address of application B's subclass function, SetClassLong returns the address of application A's subclass function instead of the address of the original window procedure for the window class. If application A removes the subclass from the edit control class, it replaces the original window procedure's address in the WNDCLASS structure. From this point on, application B's subclass is effectively removed. New edit controls have the original window procedure's address as their window procedure. If application B then removes its subclass of the edit control class, even more damaging events occur. Application B replaces the original window procedure's address in the WNDCLASS structure with the address of application A's subclass function. New edit controls now attempt to use application A's subclass function. If application A is still loaded, its subclass of the edit control class has effectively been reinstalled, even though application A is not aware of this. If application A is no longer loaded, Windows version 3.0 still attempts to call a procedure at the address of application A's subclass function; this may cause a FatalExit, a UAE, or the system to hang. Windows version 3.1 does not allow this to happen.

Because of the dangers of globally subclassing the control classes, global subclassing should be done only on application-specific window classes. If your application could benefit from globally subclassing a control class, your application should use an alternative technique called superclassing.

SUPERCLASSING

Subclassing a window class causes messages meant for the window procedure to be sent to the class function. The class function then passes the message to the original window procedure. Superclassing creates a new window class. The new window class uses the window procedure from an existing class to give the new class the functionality of the existing class. The superclass is based on some other window class, known as the base class. Frequently the base class is a Windows-supplied control class, but it can be any window class. Do not superclass the scroll bar control class because Windows uses the class name to produce the correct behavior for scroll bars.

The superclass has its own window procedure, the superclass procedure, which can take the same actions a subclass procedure can. The superclass procedure can take three actions with the message: (1) pass the message directly to the original window procedure; (2) modify the message before passing it to the original window procedure; (3) not pass the message. The superclass can react to the message before, after, or both before and after passing the message to the original window procedure.

Unlike a subclass procedure, a superclass procedure receives create (WM_NCCREATE, WM_CREATE, and so on) messages from Windows. The superclass procedure can process these messages, but it must also pass these messages to the original base-class window procedure so that the base-class window procedure can initialize. The application calls GetClassInfo to base a superclass on a base class. GetClassInfo fills a WNDCLASS structure with the values from the base class's WNDCLASS structure. The application that is superclassing the base class then sets the hInstance field of the WNDCLASS structure to the instance handle of the application. The application must also set the WNDCLASS structure's lpszClassName field to the name it wants to give this superclass. If the base class has a menu, the application superclassing the base class must supply a new menu that has the same menu IDs as the base class's menu. If the superclass intends to process the WM_COMMAND message and not pass the message to the base class's window procedure, the menu does not have to have corresponding IDs. GetClassInfo does not return the lpszMenuName, lpszClassName, or hInstance field of the WNDCLASS structure.

The last field that must be set in the superclass's WNDCLASS structure is the lpfnWndProc field. GetClassInfo fills this field with the original class window procedure. The application must save this address so that it can pass messages to the original window procedure with a call to CallWindowProc. The application must put the address of its subclass function into the WNDCLASS structure. This address is not a procedure-instance address because RegisterClass gets the procedure-instance address. The application can modify any other fields in the WNDCLASS structure to suit the application's needs.

The application can add to both the extra class bytes and the extra window bytes because it is registering a new class. The application must follow two rules when doing this: (1) The original extra bytes for both the class and the window must not be touched by the superclass for the same reasons that an instance subclass or a global subclass should not touch these extra bytes; (2) if the application adds extra bytes to either the class or the window instance for the application's own use, it must always reference these extra bytes relative to the number of extra bytes used by the original base class. Because the number of bytes used by the base class may be different from one version of the base class to the next, the starting offset for the superclass's own extra bytes is also different from one version of the base class to the next.

After the WNDCLASS structure is filled, the application calls RegisterClass to register the new window class. Windows of this class can now be created and used.

Additional reference words: 3.1 3.10

KBCategory:
KBSubcategory: UsrWinSubclass UsrCtlSubclass

INF: EM_SETSEL wParam Not Used in Single Line Edit Controls

Article ID: Q102641

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

For an edit control, the EM_SETSEL documentation states that wParam is a flag for specifying whether to scroll the caret into view. When the parameter is zero, the caret is scrolled into view. When the parameter is one (1), the caret is not scrolled into view. This is only valid for a multiline edit control. For a single line edit control, the wParam flag is not used.

Additional reference words: 3.10 EM_SETSEL
KBCategory:
KBSubcategory: UsrCtlEdit

SAMPLE: Moving an Item in a List Box Using Drag and Drop

Article ID: Q103318

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows,
   version 3.1

MOVELST demonstrates how to reorder the items in a list box using drag and drop. The user clicks an item, drags it to the new position, and drops the item. The list box will scroll if necessary.

MOVELST also shows how to implement a dialog box as a main window using a private dialog class.

The drag and drop implementation is based on the sample application LISTDRAG.

MOVELST can be found in the Software/Data Library by searching on the word MOVELST, the Q number of this article, or S14285. MOVELST was archived using the PKware file-compression utility.

Additional reference words: 3.10 listbox
KBCategory:
KBSubcategory: UsrCtlListbox

SAMPLE: Subclassing VBX Controls with MFC 2.0

Article ID: Q103856

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows
  version 3.1

SUMMARY
=======

SUBVBX is a sample application that does Windows subclassing on a VBX control using the Microsoft Foundation Class (MFC) Libraries version 2.0. The sample subclasses the grid control, overrides the OnDlgCode() message handler, and returns the DLGS_WANTARROWS code. Normally Windows uses the arrow keys to move between controls in a dialog box, and does not pass the arrow keys to the control. Subclassing the grid control and overriding the OnDlgCode() message handler causes Windows to pass the arrow keys to the grid control so that it can use the keys to move between cells in the control.

SUBVBX can be found in the Software/Data Library by searching on the word SUBVBX, the Q number of this article, or S14291. SUBVBX was archived using the PKware file-compression utility.

MORE INFORMATION
================

Windows controls, such as an edit control or a list box control, can be subclassed in MFC using the CWnd::SubclassWindow() and CWnd::SubclassDlgItem() functions. These functions do not work for VBX controls. These functions rely on the fact that each window control has its own Windows procedure. This way it is possible for an MFC object to chain to the control's original Windows procedure. However, Visual Basic (VB) controls under MFC 2.0 are managed by MFC objects, and therefore VB controls use the same Windows procedure as all other MFC objects.

To subclass a VB control in MFC 2.0, it is necessary to copy the data from the original control object into the object that you want to use to subclass the original control. After the original object has been copied, it can be detached and deleted, and the new control object can be attached.

This sample defines the CVBClone class that contains the function SubclassVBControl(). This function does the copying, attaching, and detaching that is described above. To use this class, derive a new class from the CVBClone class and new message handling functions to the message map. This class can then be used to subclass a VB control in a dialog box by calling the SubclassVBControl() function in the OnInitDialog() or OnInitialUpdate() function of the dialog box or form view that contains the control.

This sample subclasses a grid control in a form view. The sample defines the CMyGrid class from CVBControl, which it uses to subclass the control. The sample also calls SubclassVBControl in the OnInitialUpdate() of the form view to subclass the control in the form view.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrCtlSubclass

INF: SetParent and Control Notifications

Article ID: Q104069

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1

An edit, list box, or combo box control sends notifications to the original parent window even after SetParent has been used to change the control's parent. A button control sends notifications to the new parent after SetParent has been used to change its parent.

Edit, list box, and combo box controls keep a private copy of the window handle of the parent at the time of creation. This handle is not changed when SetParent is used to change the control's parent. Consequently, the notifications (EN_*, LBN_*, and CBN_* notifications) go to the original parent.

Note that WM_PARENTNOTIFY messages go to the new parent and GetParent() returns the new parent. If it is required that notifications go to the new parent window, code must be added to the old parent's window procedure to pass on the notifications to the new parent.

For example:

```
case WM_COMMAND:
  hwndCtl = LOWORD(lParam);

  // If notification is from a control and the control is no longer this
  //   window's child, pass it on to the new parent.
  if (hwndCtl && !IsChild(hWnd, hwndCtl))
      SendMessage(GetParent(hwndCtl), WM_COMMAND, wParam,  lParam);
  else Do normal processing;
```

Button controls send notifications to the new parent after SetParent has been used to change the parent.

Additional reference words: 3.10 listbox combobox
KBCategory:
KBSubcategory: UsrCtl

INF:Unselecting Edit Control Text at Dialog Box Initialization

Article ID: Q96674

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

To remove the highlight (selection) from an edit control text, an EM_SETSEL message must be sent to the control. However, while processing the WM_INITDIALOG message of a dialog box, sending an EM_SETSEL fails to remove the highlight from (unselect) the edit control text.

This article describes a method to work around this behavior.

More Information:

When a newly created dialog box is displayed with focus on an edit control, the default text of the edit control is shown highlighted. In some cases, the text highlighting is undesirable because accidentally pressing a character key removes the original text from the edit control. Therefore, the workaround is to unselect the text by sending an EM_SETSEL message to the edit control at the dialog box initialization.

While processing the WM_INITDIALOG message, sending the EM_SETSEL message fails to remove the highlight from the edit control. This happens because the edit control has not yet been drawn. Because it's not drawn and there is no selection information available to the edit control's procedure, the EM_SETSEL message is ignored. In other words, the SendMessage function passes the EM_SETSEL message too early to the edit control for it to become effective.

To work around this behavior, delay the EM_SETSEL message until the focus is set to the edit control. That is, while processing the first EN_SETFOCUS notification message, an EM_SETSEL message must be sent to the edit control to remove the highlight from its text. For example:

```
    static  BOOL    bFirstTime;   // We want to unselect only once.

    switch ( message )
    {
        case WM_INITDIALOG:
            bFirstTime = TRUE;
            return TRUE;

        case WM_COMMAND:
            switch ( wParam )
            {
                case IDC_EDIT:
          // If this is the first time, then unselect.
                    if ( HIWORD( lParam ) == EN_SETFOCUS &&
```

```
                 bFirstTime )
                     {
                         SendMessage( GetDlgItem( hwndDialog, IDC_EDIT ),
                                                                 EM_SETSEL, 0,
MAKELPARAM( -1, -1 ));
                         bFirstTime = FALSE;
                     }
                     break;
         .
         .
         .
             }  // switch ( wParam )
     .
     .
     .
     } // switch ( message )
```

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrCtlEdit

INF: MAZE Program from

Article ID: Q32931

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

MAZE is a file in the Software/Data Library that contains the source code to an application that demonstrates how to use dynamic data exchange (DDE) in a Windows application. MAZE can be found in the Software/Data Library by searching on the word MAZE, the Q number of this article, or S10007. MAZE was archived using the PKware file-compression utility.

When multiple instances of the MAZE application are started, MAZE draws a bouncing ball in the client area of one of its instances. MAZE creates a DDE communications link between instances of the MAZE application and uses this link to pass the ball to other instances at random. MAZE also creates a DDE conversation with Excel and passes statistical information about the ball to Excel.

MAZE demonstrates how to use DDE to pass information between applications and how to use PeekMessage to allow other applications to run simultaneously with a process-intensive task.

Additional reference words: 2.03 3.00 softlib 2.x
KBCategory:
KBSubcategory: UsrDdeProtocol

PRB: Windows REQUEST Function Not Working With Excel

Article ID: Q26234

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

The REQUEST function does not work correctly with Excel. The request message is received, however, Excel does not process the WM_DDE_DATA message that is sent back.

RESOLUTION

The fResponse bit must also be set in the WM_DDE_DATA message (bit 12). This bit tells Excel that the data message is in reply to a REQUEST function and not an ADVISE function. If "lpddeup->fResponse=1" is added, the REQUEST function should work correctly.

Additional reference words: TAR68222 2.x 2.03 2.10 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrDdeExcelww

Windows SDK: DDE Protocol for Initiate and Acknowledge
Article ID: Q69028

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article discusses the dynamic data exchange (DDE) protocol outlined in the Windows Software Development Kit (SDK) manuals. In particular, the article describes the proper handling of atoms during processing of the WM_DDE_INITIATE and WM_DDE_ACK messages.

More Information:

These code samples are provided in Chapter 22 of the "Microsoft Windows Software Development Kit Guide to Programming" as examples of the proper way to initiate a DDE conversation.

This code from page 22-7 demonstrates how a DDE client initiates a conversation:

```
   atomApplication = GlobalAddAtom("Server");
   atomTopic       = GlobalAddAtom(szTopic);

   SendMessage(-1, WM_DDE_INITIATE, hwndClientDDE,
           MAKELONG(atomApplication, atomTopic)) )

   GlobalDeleteAtom(atomApplication);
   GlobalDeleteAtom(atomTopic);
```

In response to this message, any active DDE server(s) is to create an atom with the name of each topic that it supports and use SendMessage to SEND a WM_DDE_ACK message back to the client.

This code is taken from pages 22-8 and 22-9:

```
   atomApplication = GlobalAddAtom("Server");
   atomTopic       = GlobalAddAtom(szTopic);
   if (!SendMessage(hwndClient, WM_DDE_ACK, hwndServerDDE,
           MAKELONG(atomApplication, atomTopic)))
   {
       GlobalDeleteAtom(atomApplication);
       GlobalDeleteAtom(atomTopic);
   }
```

The "if" statement here is unnecessary because the client will delete the atoms sent to it during the processing of the WM_DDE_ACK message. This code should be rewritten as:

```
   atomApplication = GlobalAddAtom("Server");
   atomTopic       = GlobalAddAtom(szTopic);
```

```
   // Check to see if we support topic
   if (MAKELONG(atomApplication, atomTopic) == lParam)
   {
       // create server window
       hWndServer = CreateServerWnd(...);

       // acknowledge conversation
       SendMessage(hwndClient, WM_DDE_ACK, hwndServer,
               MAKELONG(atomApplication, atomTopic));
   }
   else  // if not then delete the atoms WE created
   {
       GlobalDeleteAtom(atomApplication);
       GlobalDeleteAtom(atomTopic);
   }
```

This code demonstrates how the client should process the WM_DDE_ACK message:

```
   case WM_DDE_ACK:
       if (fInitiate)         /* In initiate sequence */
       {
        /* take action */
        GlobalDeleteAtom(LOWORD(lParam));
        GlobalDeleteAtom(HIWORD(lParam));
       }
       /* respond to other type of ACKs */
       return 0;
```

There is some confusion as to what exactly should be done in response to a WM_DDE_ACK message. Page 15-8 of the "Microsoft Windows Software Development Kit Reference Volume 2" specifies four things to check when responding to a WM_DDE_ACK message:

1. Delete all atoms associated with the ACK message.

2. If the WM_DDE_ACK has an accompanying hData object, the object should be deleted.

3. If the WM_DDE_ACK is a negative response to a WM_DDE_ADVISE, the hOptions object sent with the WM_DDE_ADVISE message should be deleted.

4. If the WM_DDE_ACK is a negative response to a WM_DDE_EXECUTE, the hCommands object sent with the WM_DDE_EXECUTE message should be deleted.

Since the sending application has no way of telling when the WM_DDE_ACK is received, it is important that the receiver perform the necessary cleanup upon receipt of the message.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeProtocol

INF: Initiating DDE Conversation w/ Instance of Windows Excel

Article ID: Q29547

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The documentation for the EXEC() macro on page 70 of the "Microsoft Excel Function Reference" manual for version 3.0 incorrectly refers to its return value as "the Microsoft Windows task ID number of the started program." However, it is not the value returned by the Windows KERNEL routine GetCurrentTask(). EXEC() returns the instance handle to the EXEC()'d application.

More Information:

Windows and Excel do not provide an easy way to find out which instance of Excel called the application. However, the correct value can be obtained.

The following is a sample macro that uses the REGISTER and CALL functions to obtain the instance handle to the version of Excel that is running. That information then can be passed to the application inside the EXEC call. However, the value in hInstance must be appended as a string for the EXEC call.

The following is the text version of the macro sheet:

```
-------------------------------------------------------------
             A                                 |        B
This is an example of how to obtain the |
instance and then pass that value to the|
EXEC'd application:                     |
                                        |
=STEP()                                 |
Get instance handle of current window.  |
=REGISTER("USER","GetFocus","H")"       |
=CALL(A7)                               | <== hWnd
       -6                               | <==GWW_HINSTANCE
=REGISTER("user","GetWindowWord","HHI")"| <==GetInstance
=CALL(GetInstance,hWnd,GWW_HINSTANCE)"  | <== hInstance
=RETURN()                               |
Exec("YourApp.exe  hInstance",1)"       |
-------------------------------------------------------------
```

Note: For Microsoft Excel versions 2.x, the corresponding documentation for the EXEC() macro is on page 275 of the "Microsoft Excel Functions and Macros" manual.

Additional reference words: 3.00
KBCategory:

KBSubcategory: UsrDdeExcelww

INF: DDE Sample Code in Software Library

Article ID: Q73681

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

DDE.ZIP is a code sample in the Software/Data library that demonstrates dynamic-data exchange (DDE) and how it is programmed under Windows 3.0.

DDE can be found in the Software/Data Library by searching on the keyword DDE, the Q number of this article, or S10036. DDE was archived using the PKware file-compression utility.

More Information:

There are two separate applications in the DDE archive file: TICTAPE and BARGRAPH. For each application, the archive holds a makefile, header file, C source code, resources (RC) file, segment definitions (DEF) file, and an icon.

TICTAPE is a DDE server that randomly creates and transmits data about four stocks. BARGRAPH is a DDE client that graphically illustrates the data from TICTAPE. TICTAPE was demonstrated at "Windows on Wall Street," June 4, 1986.

To use these applications, perform the following six steps:

1. Download from the Software/Data Library and expand the DDE.ZIP file.

2. Build TICTAPE and BARGRAPH.

3. Start Windows.

4. Run TICTAPE.EXE.

5. Run BARGRAPH.EXE.

6. Activate TICTAPE and choose "Start Ticker Tape".

The BARGRAPH application's main window will dynamically display the (random) values of the four stocks.

Additional information concerning DDE can be found in Chapter 22 of the "Microsoft Windows Software Development Kit Guide to Programming" and in Chapter 15 of the "Microsoft Windows Software Development Kit Reference, Volume 2."

Additional reference words: 3.00
KBCategory:

KBSubcategory: UsrDdeRaremisc

Modifying the SDK DDE Server Example to Work with Excel
Article ID: Q69293

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The sample dynamic data exchange (DDE) application provided in the Microsoft Windows Software Development Kit (SDK) does not work properly with Microsoft Excel when Excel makes a DDE request (=REQUEST). The information below discusses the necessary changes to the Server application to achieve compatibility with Excel.

More Information:

Excel uses the fResponse flag in the DDEDATA structure to distinguish responses from WM_DDE_REQUEST and WM_DDE_ADVISE messages. If a DDE server application responds to a WM_DDE_REQUEST message from Excel, it must set the fResponse flag to TRUE. If a DDE server application responds to a WM_DDE_REQUEST with the fResponse flag set to FALSE, the Excel macro will wait until interrupted by the user.

The modifications to the Server application required for Excel compatibility are described below.

The SendData() function in the sample application should be changed to look like the following:

```
   void SendData(HWND hwndServerDDE,
                 HWND hwndClientDDE,
                 char *szItemName,
                 char *szItemValue,
                 BOOL bDeferUpdate,
                 BOOL bAckRequest,
                 BOOL bAdviseResponse)
```

The new argument bAdviseResponse is set to TRUE when the data block to be sent is a response to a WM_DDE_ADVISE message; otherwise, it is set to FALSE.

After the data block is allocated, the response flag is set using:

```
   lpData -> fResponse = !bAdviseResponse ;
```

These two changes are made in the SERVDDE.C file of the sample. All other references to SendData() in SERVER.C, SERVER.H, or SERVDDE.C should be modified to account for the new argument.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeExcelww

INF: Passing METAFILEPICT Structures Through DDE
Article ID: Q69883

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When an application sends a WM_DDE_REQUEST message and the server application replies with a WM_DDE_DATA message, the format of the data returned in the Value[] member of the DDEDATA structure is not always the same, but depends on the value of the cfFormat member of the structure.

More Information:

The server application defines the data returned in the Value[] member of the DDEDATA structure. This data must be in one of the formats used to pass data to the Clipboard. The standard clipboard data formats and a description of their data are defined in the documentation for the SetClipboardData() function on pages 4-370 and 4-371 in the "Microsoft Windows Software Development Kit Reference Volume 1". For example, for the CF_TEXT format, the actual data is returned from the DDE server, and for the CF_BITMAP format, a handle to the bitmap is sent.

The table is not explicit as to what data is returned for the CF_METAFILEPICT data format. The Value[] parameter contains a handle to a global memory block containing a METAFILEPICT structure. To access this structure, call GlobalLock() to get a pointer to the memory and cast the pointer to LPMETAFILEPICT. For example:

```
  LPMETAFILEPICT  lpMFP;

  lpMFP = (LPMETAFILEPICT)GlobalLock(lpDDEData->Value);
```

At this stage, the metafile bits may be copied with the following statement:

```
  hMFBits = CopyMetaFile(lpMFP->hMF, NULL);
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeRaremisc

Sources of Information Regarding Windows DDE

Article ID: Q69889

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The information below contains a list of references on Windows dynamic data exchange (DDE) and some tools that can be used to test DDE applications developed with the Windows Software Development Kit (SDK) version 3.00.

More Information:

PRINTED DOCUMENTATION
=====================

DDE Specification

- "Microsoft Windows Software Development Kit Reference Volume 2," Chapter 15, "Windows DDE Protocol Definition."

- "Microsoft Windows Software Development Kit Guide to Programming," Chapter 22.

Books

- "Programming Windows," Second Edition (Charles Petzold, Microsoft Press), Chapter 17.

Microsoft Systems Journal Articles

- Vol. 5 No. 1 (January 1990), "Simplifying Complex Windows Development Through the Use of a Client-Server Interface."

- Vol. 4 No. 3 (May 1989), "A Technical Study of Dynamic Data Exchange Under OS/2 Presentation Manager."

Sample Code

- Windows 3.00 SDK DDE client and server applications (in the \SAMPLES\DDE directory)

- DDEPOP and SHOWPOP sample applications in "Programming Windows."

COMMERCIAL APPLICATIONS THAT USE DDE
====================================

Below is a list of commercial applications that support DDE and the conversation topics that each supports.

Microsoft Excel

See the index of the "Microsoft Excel User's Guide" for complete references.

Application	DDE Topics	Items
NULL	NULL	
Excel	Sheet name	Cell reference (must be in R1C1 format)
Excel	System	SysItems
Excel	System	Formats (eight different kinds)
Excel	System	Topics (sheets that are loaded)
Excel	System	Status (busy or not)

Q+E

See pages 74-79 of the Q+E manual shipped with Microsoft Excel 3.00 for more information.

Application	DDE Topics	Items
NULL	NULL	
QE	System	SysItems
QE	System	Formats = Biff TEXT
QE	System	Topics = Query1
QE	System	Status = Ready
QE	System	Logon
QE	System	Logoff
QE	System	Sources
QE	System	Tables

Microsoft Word for Windows

Please see the "Microsoft Word for Windows Technical Reference" manual (published by Microsoft Press) for more information.

Application	DDE Topics	Items
NULL	NULL	
WinWord	Document	bookmark
WinWord	System	SysItems
WinWord	System	Formats (text, rtf, metafilepict, bitmap, link)
WinWord	System	Topics (documents that are loaded)

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeOtherapps

PRB: ExitProgman DDE Service Does Not Work If PROGMAN Is Shell
Article ID: Q69899

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

SYMPTOMS
   Calling the ExitProgman() function documented in the "Microsoft
   Windows Software Development Kit Guide to Programming," section
   22.4.4 (pages 22-19 through 22-22) fails under certain
   circumstances.

CAUSE
   Calling this function will fail if the Program Manager is the
   Windows shell.

RESOLUTION
   For this function to work correctly, another shell program must be
   specified in the SYSTEM.INI file and the Program Manager must be
   run from that shell.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeProgman

INF: Using the SDK CLIENT Sample with Commercial Applications
Article ID: Q71225

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When the Client application included with the Windows Software Development Kit (SDK) version 3.00 is used with Word for Windows or with Excel, the System topic is available but no documents. This problem is caused by the limit that the Client application has placed on the length of the strings used in DDE conversations.

More information:

The length of the strings the Client and Server sample applications use is limited to 8 characters.

To correct this limitation, it is necessary to modify CLIENT.H and SERVER.H. The symbolic constants listed below must be #defined to have values larger than 8; 14 was chosen as an example. After modifying these variables the applications must be rebuilt.

```
#define APP_MAX_SIZE             14
#define TOPIC_MAX_SIZE           14
#define ITEM_MAX_SIZE            14
#define VALUE_MAX_SIZE           14
#define EXECUTE_STRING_MAX_SIZE  100
#define CONV_MAX_COUNT           14
#define ITEMS_PER_CONV_MAX_COUNT 5
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeExcelww

INF: Program Manager DDE Command AddItem Documentation
Article ID: Q86872

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The Microsoft Windows Program Manager has a dynamic data exchange (DDE) command-string interface that allows another application to create, update, and delete groups and programs within groups. The text below extends the documentation for the AddItem command that begins on page 373 of the "Microsoft Windows Software Development Kit: Reference, Volume 1: Overview."

More Information:

The syntax for the AddItem command has the following form:

```
   AddItem(CmdLine[, Name[, IconPath[, IconIndex[, xPos, yPos[,
           DefDir[, HotKey[, fMinimize] ] ] ] ] ])
```

The AddItem command instructs Program Manager to add an icon to an existing group.

If you specify fewer than four parameters, the IconIndex defaults to the last icon in the group. For example:

```
   AddItem(notepad.exe, "Notepad", notepad.exe)
```

If you specify five or more parameters but do not specify values for xPos and yPos, the icon position defaults to 0, 0 (the first icon in the group). For example:

```
   AddItem(notepad.exe, "Notepad", notepad.exe,,,, "c:\")
```

Therefore, to create the last icon in the group and specify five or more parameters, set xPos and yPos to -1. For example:

```
   AddItem(notepad.exe, "Notepad", notepad.exe,, -1, -1, "c:\")
```

For the HotKey parameter, specify the ASCII value of the desired key. The application can create a key combination (for example, ALT+SHIFT+A) by adding one or more of the following values to the ASCII value for the key:

Key	Decimal Offset
SHIFT	256
CTRL	512
ALT	1024

Extended   2048

For example, ALT+SHIFT+A is signified by 1345 (1024+256+65).

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDdeProgman

INF: Drawing the Icon for a Minimized Application

Article ID: Q74539

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The text below describes how an application developed for the Microsoft Windows graphical environment can draw its own icon when it is minimized. An application can use these techniques to draw different icons to indicate its state.

In Windows version 3.1, an application can initiate a dynamic data exchange (DDE) conversation with the Windows Program Manager to determine which icon the user has associated with an application and can then draw that icon using the following techniques.

More Information:

To draw its own icon when minimized, a Windows application performs the following five steps:

1. When the application registers its main window's window class, specify NULL as the value of the hIcon member of the WNDCLASS data structure.

2. Create a static variable to store the icon the application will use, as follows:

   ```
   HICON hIcon;
   ```

   Initialize this variable to the value returned from the CreateIcon or LoadIcon function.

3. To draw the icon, process the WM_PAINT message in the main window's window procedure as follows:

   ```
   case WM_PAINT:
      {
      PAINTSTRUCT ps;

      if (IsIconic(hWnd))
         {
         BeginPaint(hWnd, (LPPAINTSTRUCT)&ps);

         // Paint the desktop window background.
         DefWindowProc(hWnd, WM_ICONERASEBKGND, (WORD)ps.hdc, 0L);

         // Draw the icon on top of it.
         DrawIcon(ps.hdc, 0,0, hIcon);
   ```

```
         EndPaint(hWnd, (LPPAINTSTRUCT)&ps);
         }
      else
         return DefWindowProc(hWnd, message, wParam, lParam);
      }
```

4. To prevent a screen flash when the application draws the icon, process the WM_ERASEBKGND message in the main window's window procedure, as follows:

```
   case WM_ERASEBKGND:
      if (IsIconic(hWnd))
         // Do not erase the background now. When the application
         // paints its icon, it will erase the background.
         return TRUE;
      else
         return DefWindowProc(hWnd, message, wParam, lParam);
```

5. Process the WM_QUERYDRAGICON message in the main window's window procedure. When the user is about to drag a window that has no icon defined for its window class, Windows sends it a WM_QUERYDRAGICON message. The code below causes Windows to create a dragging cursor based on the specified icon. Windows automatically converts a color icon to a monochrome cursor.

```
   case WM_QUERYDRAGICON:
      return (LONG)(WORD)hIcon;
      break;
```

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrDdeProgman

INF: Communicating Between Windows Virtual Machines with DDE

Article ID: Q40620

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Windows 286 supports DDE (dynamic data exchange) for DOS (non-Windows) applications; however, Windows/386 and Windows versions 3.x real and enhanced modes do not include this type of functionality. Currently, there are no plans to add this type of functionality to a future release of Windows.

The following methods describe several different ways for a DOS (non-Windows) application to communicate with a Windows/386 application:

1. The Clipboard. DOS applications can read from or write to the Clipboard. The methods for doing this are documented in the Windows SDK (Software Development Kit). Microsoft Word incorporates this capability. This would be the method of choice in most instances.

2. A shared file. This is not an extremely fast method, although the programmer could use a file on a RAM disk or rely on SMARTDRV to reduce actual file accesses. Remember to call SHARE.EXE from DOS before starting Windows/386 to ensure that file sharing works properly.

3. DOS TSR (terminate-and-stay-resident) program. Another method that can be used is to start a DOS TSR program with a buffer at a fixed location that can be read/written to by different applications. For this method to work, the memory region used must be below the load point of Windows/386, otherwise the different virtual machines will have their own private copies of the address space.

4. Bridge/386. There is a program called Bridge/386 that is sold by Softbridge Microsystems in Cambridge, Massachusetts, that provides this type of functionality. There is a good description of this program in the September 1988 issue of the "Microsoft Systems Journal."

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeOtherapps

INF: How to Get a List of Program Manager Groups with DDE

Article ID: Q74858

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

An application can obtain a list of groups from the Program Manager by making a WM_DDE_REQUEST for the "PROGMAN" item. The groups are returned in CF_TEXT format as a list of strings separated by carriage returns and terminated with a NULL. For example:

```
Main\x0d\x0aAccessories\x0d\x0a\x00
```


The procedure is detailed in Section 22.4.2 of the "Microsoft Windows Software Development Kit Guide to Programming" in the "Obtaining an Item from the Server" subsection. The application should use "PROGMAN" as the szItemName and it should make a local copy of the data as outlined on page 22-11.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDdeProgman

INF: Is DdePostAdvise Synchronous?

Article ID: Q92540

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The server's callback function will have received XTYP_ADVREQ for each existing advise loop that was started without a XTYPF_ACKREQ flag, by the time DdePostAdvise returns. So DdePostAdvise is synchronous except for advise loops started with the XTYPF_ACKREQ flag.

More Information:

The DdePostAdvise function causes the Dynamic Data Exchange Management Library (DDEML) to send a XTYP_ADVREQ transaction to the calling (server) application's DDEML callback function for each client that has an advise loop active on the specified topic or item name pair. A server application calls this function whenever the data associated with the topic or item name pair changes.

DDEML will have sent XTYP_ADVREQ for each advise loop that was started without a XTYPF_ACKREQ when DdePostAdvise returns.

If an advise loop was started with the XTYPF_ACKREQ flag, the server will have to wait until the client acknowledges that it received the previous data item. Consequently, if the client did not acknowledge the previous data item, DdePostAdvise will set a flag for the advise loop indicating the data change and XTYP_ADVREQ will be sent by DDEML only after the client acknowledges the previous data item. The cAdvReq parameter of XTYP_ADVREQ for the slow client will contain CADV_LATEACK instead of the remaining advise loops to be processed.

This means that DdePostAdvise can return before the server callback receives the XTYP_ADVREQ corresponding to a XTYPF_ACKREQ advise loop. This is not a problem because when the client used XTYPF_ACKREQ, it was willing to miss some of the data transitions in order to prevent the server from sending it data faster than could be processed. The server can send the slow client the latest data when it finally receives the XTYPF_ADVREQ with cAdvReq set to CADV_LATEACK.

Additional reference words: 3.00 3.0 3.10 3.1 call back
KBCategory:
KBSubcategory: UsrDdeProtocol

INF: DDEDATA Documentation Error
Article ID: Q93372

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Page 272 of the Windows SDK (Software Development Kit) version 3.1 "Programmer's Reference, Volume 3: Messages, Structures, and Macros" contains an error. The fAckReq field of the DDEDATA structure describes what should be the fResponse field, and vice versa.

In addition, the DDEDATA structure documented on page 15-11 of the Windows SDK version 3.0 "Reference, Volume 2" incorrectly describes what should be the fResponse field, as fRequested.

More Information:

The Windows SDK versions 3.0 and the 3.1 both define the DDEDATA structure in DDE.H as follows:

```
     typedef struct tagDDEDATA
     {
         WORD  unused: 12,
                          fResponse:1,
                          fRelease:1,
                          reserved:1,
                          fAckReq:1;
         int            cfFormat;
         BYTE     value[1];
     } DDEDATA;
```

The WM_DDE_DATA documentation for Windows SDK versions 3.0 and 3.1 correctly describes the structure's fAckReq field. fAckReq is a flag to indicate whether the server application expects to be acknowledged by the client application (with a WM_DDE_ACK) when data is received.

The fResponse field, on the other hand, is just a means to distinguish whether the WM_DDE_DATA message the client received was sent in response to a WM_DDE_REQUEST (if it is, it is set to TRUE) or WM_DDE_ADVISE (in which case it is set to FALSE).

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrDdeRaremisc

INF: How to Use Network DDE

Article ID: Q100363

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

Summary:

With Windows for Workgroups, an application can use Dynamic Data Exchange (DDE) over a network (NetDDE).

When using normal DDE or DDEML in an application, the various fields were set up as follows (using Excel as an example):

```
  APPLICATION       TOPIC     ITEM
  Excel             System     Topics
```

However, to start a DDE conversation over the network, a client application must establish the DDE conversation with another machine by specifying the following strings for the application name and topic

```
   application:   \\machine-name\NDDE$
   topic:           share-name
```

where the share name is defined in the [DDEShare] section of SYSTEM.INI. The default installation of Windows for Workgroups contains the following DDEShares:

```
   [DDEShares]
   HEARTS$=mshearts,hearts,,15,,0,,0,0,0
   CLPBK$=clipsrv,system,,31,,0,,0,0,0
   CHAT$=winchat,chat,,31,,0,,0,0,0
```

More Information:

To illustrate how to use NetDDE, ClipBook will serve as a server example.

The following are specifics about ClipBook:

```
   Title Bar Caption     = ClipBook Viewer,
   Executable File Name = Clipbrd.exe,
   DDE Application Name = clipsrv,
   DDE Share Name        = CLPBK$
```

To start a conversation with ClipBook from your application, use the following strings:

```
   APPLICATION              TOPIC        ITEM
   \\machine-name\NDDE$    CLPBK$     Topics
```

Under normal DDE, this would be viewed as:

```
SERVER NAME       APPLICATION       TOPIC     ITEM
\\machine-name    Clipsrv           System     Topics
```

This will return a list of SHARED "objects" in the ClipBook. Now that you have a list of objects, the next step is to query the types of formats available. For example, assume that "$Test" was returned as an object. To find a list of formats, start a new conversation with ClipBook with the following parameters:

```
APPLICATION             TOPIC           ITEM
\\machine-name\NDDE$    $Test           FormatList
```

This will return a list of the formats, each starting with the "&" symbol (for example, CF_TEXT will return "&Text").

Using the object name and format type, it is possible to make a connection to the server and return the contents of the topic. The last set of parameters is:

```
APPLICATION             TOPIC           ITEM
\\machine-name\NDDE$    $Test           &Text
```

If your application wants to share information via NetDDE, use the ClipBook program or call one of the NetDDE application programming interfaces (APIs), which will create an entry in the [DDEShares] section of SYSTEM.INI. This procedure allows other users to access your information.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDdeRaremisc

INF: Using ReplaceItem() Command in Program Manager DDE
Article ID: Q102590

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

Program Manager has a dynamic-data exchange (DDE) command-string interface that allows other applications to manipulate its group windows. Windows version 3.1 has a new command called ReplaceItem(), which instructs Program Manager to delete an item and replace it with a new item when Program Manager receives a subsequent AddItem() command.

MORE INFORMATION
================

The syntax for the ReplaceItem() command is

```
   ReplaceItem(ItemName)
```

where ItemName is the name of the item that is to be replaced.

The ReplaceItem() command instructs Program Manager to delete the given ItemName and record the position of the deleted item. The next time Program Manager receives an AddItem() command, it adds the new item to this previously recorded position, effectively replacing the deleted item with the new one.

For example, Program Manager has a group called "MAIN" with an item called "TEST". To replace the item TEST with a new item "NEW", the following commands must be executed:

```
   ShowGroup ("MAIN", 1);    //Activate the group first.
   ReplaceItem ("TEST");
   AddItem ("NEW");
```

The position of the newly added item will be the same as the deleted item (TEST) regardless of what parameters are passed in the AddItem() command. Given that the position (xpos, ypos) of item TEST was (10, 10) in the group MAIN before it was replaced by item NEW, specifying a new (xPos, yPos) position for NEW in the AddItem() command has no effect. Program Manager adds the NEW item at the recorded position (10, 10) in MAIN.

Additional reference words: 3.10 progman
KBCategory:
KBSubcategory: UsrDdeProgman

PRB: AddAtom Causes Divide by Zero Error

Article ID: Q103036

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows,
   version 3.1

SYMPTOMS
========

Calling the AddAtom function in the Windows 3.x environment with a string that begins with the pound (#) character causes the system to display the following error message in a system modal dialog:

   Application Error

   integer divide by 0

This error occurs if the first AddAtom call in the application passes a string containing a # as the first character, and passes non-numeric characters in the rest of the string. For example, the first call to AddAtom with the string "#string" will cause the error to occur.

RESOLUTION
==========

If it is necessary to have atom strings beginning with the # character, first call AddAtom with a false string that contains no # characters. Alternatively, call InitAtomTable before adding the first atom string beginning with the # character.

STATUS
======

Microsoft has confirmed this to be a problem in Windows versions 3.0 and 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

Atoms with names begin with the # character are strongly discouraged. The # character serves a special purpose for atoms. If the string that is passed to AddAtom has the form "#1234", AddAtom returns an integer atom whose value is the 16-bit value representation of the decimal number specified in the string. If the decimal value specified is 0 (zero) or a value in the range 0xC000 through 0xFFFF, the return value is zero, indicating an error.

Sample Code

```
/* Compile options needed:  /Zp /GA or /GD or /Gw
```

```
*/

// errant code
{
 ATOM at;

 at = AddAtom("#string");     // causes error to occur

}

// corrected code
{
 ATOM at;

 InitAddTable(17);

 // or call AddAtom with a false string

 AddAtom("false");

 at = AddAtom("#string");     // no error occurs

}
```

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrDdeAtoms

INF: Dynamic Data Exchange Interface for Replacement Shells
Article ID: Q104394

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

This article is a reprint of the Windows 3.1 SDK WIN31WH.HLP section on the WinOldAp dynamic data exchange (DDE) specification.

MORE INFORMATION
================

You may choose to write an application that replaces the Windows shell. This replacement shell must be able to provide property information to the application that starts non-Windows programs in an MS-DOS window. (This application is known as WinOldApp.) This section discusses how a replacement shell can provide property information for WinOldApp. Applications other than WinOldApp do not need this information. The DDE protocol described in this section may not be supported in future versions of Windows.

Properties

A replacement shell should maintain several pieces of information, called properties, for each application that WinOldApp might start. These are the same properties that appear in the Program Item Properties dialog box of Program Manager. These properties include:

   Description (title)

   Command line

   Working directory

   Shortcut key

   Icon

The shell must provide a DDE interface that allows WinOldApp to obtain three of these properties: description, working directory, and icon. To obtain its properties from the shell, WinOldApp must accomplish the following tasks:

 - Establish a DDE conversation with the shell.

 - Request a property from the shell.

 - Receive a property from the shell.

- Terminate the DDE conversation.

Establishing a DDE Conversation

WinOldApp requests property data from the shell by using the SendMessage function to broadcast the WM_DDE_INITIATE message. The wParam parameter of the SendMessage function is the handle of WinOldApp's DDE window. The low-order word of the lParam parameter is an atom that represents the name of the shell application: "Shell". The high-order word is an atom that represents the name of the properties topic: "AppProperties". A "Shell" DDE server that supports the AppProperties topic responds to the WM_DDE_INITIATE message by sending a WM_DDE_ACK message. The server should send the following parameters with the WM_DDE_ACK message:

Parameter	Description
hwnd	Specifies the handle of WinOldApp's DDE window. The shell should use the handle that WinOldApp specified as the wParam parameter in the WM_DDE_INITIATE message.
message	Specifies the WM_DDE_ACK message.
wParam	Specifies the handle of the "Shell" server's DDE window.
HIWORD(lParam)	Specifies an atom that represents the name of the shell application: "Shell".
LOWORD(lParam)	Specifies an atom that represents the name of the properties topic: "AppProperties".

It is not necessary to free the atoms used in a conversation with WinOldApp. It is WinOldApp's responsibility to create and free the atoms.

Providing Property Data

After the DDE server that provides a replacement shell responds with a WM_DDE_ACK message to the WM_DDE_INITIATE from WinOldApp, WinOldApp sends a WM_DDE_REQUEST message to request property data. The server can respond to the WM_DDE_REQUEST message by posting a WM_DDE_DATA message.

The Windows shell associates an item name with each of the application properties that it provides. The item names are described in the following table:

Item name	Description
GetDescription	The shell provides an application's description (title) property.

GetWorkingDIR	The shell provides an application's working-directory property.
GetIcon	The shell provides an application's icon property.

WinOldApp requests properties by obtaining an atom for each of the item-name strings and passing the atoms to the shell in a sequence of WM_DDE_REQUEST messages (one message for each property). WinOldApp also passes the handle of the application's instance as the low-order word of the lParam parameter in the WM_DDE_REQUEST message. The shell should use this instance handle to find the properties associated with the application. If a "Shell" DDE server does not recognize the application's instance handle, the server does not support properties for the application instance. In this case, the server should respond by sending a negative WM_DDE_ACK message. The parameters passed with the negative WM_DDE_ACK message are as follows:

Parameter	Description
hwnd	Specifies the handle of WinOldApp's DDE window. The shell should use the handle that WinOldApp specified as the wParam parameter in the WM_DDE_REQUEST message.
message	Specifies the WM_DDE_ACK message.
wParam	Specifies the handle of the "Shell" server's DDE window.
LOWORD(lParam)	Specifies zero. The "Shell" DDE server does not support properties for the specified application instance.
HIWORD(lParam)	Specifies an atom that represents the item name of the requested property. Depending on the type of property requested, the atom should identify one of the following strings: "GetDescription", "GetWorkingDIR", or "GetIcon".

When WinOldApp receives a negative WM_DDE_ACK message, it terminates the conversation with the "Shell" DDE server. If a "Shell" DDE server recognizes the application's instance handle and the requested property is available, it should allocate a global memory object and copy the property data to the object. Then it should post a WM_DDE_DATA message to WinOldApp. The WM_DDE_DATA message should contain the handle of the global memory object.

The contents of the global memory object allocated by the shell depend on the type of property WinOldApp requested. The following three sections describe the description, working-directory, and icon properties.

Providing the Description Property

If the shell is responding to a request for the "GetDescription" property, the shell should pass the following parameters with the WM_DDE_DATA message:

```
   Parameter       Description
   -------------------------------------------------------------------

   hwnd            Specifies the handle of WinOldApp's DDE window. The
                   shell should use the handle that WinOldApp specified
                   as the wParam parameter in the WM_DDE_REQUEST message.

   message         Specifies the WM_DDE_DATA message.

   wParam          Specifies the handle of the shell's DDE window.

   LOWORD(lParam)  Specifies a handle to a global memory object that
                   contains a DDEDATA structure. A description of the
                   contents of the DDEDATA structure follows this table.
                   To report an error, the server should use one of the
                   error values listed with the WinExec function.

   HIWORD(lParam)  Specifies an atom that represents the string,
                   "GetDescription".
```

The low-order word of the lParam parameter should be a handle to a global memory object that contains a DDEDATA structure (defined in the DDE.H header file). The contents of the DDEDATA structure are as follows:

```
#include <dde.h>

typedef struct tagDDEDATA {   /* ddedat */
    WORD    unused:12,
            fResponse:1,
            fRelease:1,
            reserved:1,
            fAckReq:1;
    short   cfFormat;
    BYTE    Value[1];
} DDEDATA;
```

The Value member should contain the description property, in the form of a null-terminated string of characters from the Windows character set. The string can be any size but typically contains fewer than 30 characters. If the server sets the fAckReq bit, WinOldApp responds to the WM_DDE_DATA message by posting a WM_DDE_ACK message after processing the data. If the server sets the fRelease bit, WinOldApp frees the global memory object after copying the description string. Otherwise, WinOldApp does not free the memory object.

Providing the Working-Directory Property

If the shell is responding to WinOldApp's request for the "GetWorkingDIR" property, the shell passes the following parameters with the WM_DDE_DATA message:

```
   Parameter       Description
   ------------------------------------------------------------------

   hwnd            Specifies the handle of WinOldApp's DDE window. The
```

shell should use the handle that WinOldApp specified as the wParam parameter in the WM_DDE_REQUEST message.

message	Specifies the WM_DDE_DATA message.
wParam	Specifies the handle of the shell's DDE window.
LOWORD(lParam)	Specifies a handle to a global memory object that contains a DDEDATA structure. A description of the contents of the DDEDATA structure follows this table. To report an error, the server should use one of the error values listed with the WinExec function.
HIWORD(lParam)	Specifies an atom that represents the string, "GetWorkingDIR".

The low-order word of the lParam parameter is a handle to a global memory object that contains a DDEDATA structure. The contents of the DDEDATA structure are as follows:

```
#include <dde.h>

typedef struct tagDDEDATA {   /* ddedat */
    WORD    unused:12,
            fResponse:1,
            fRelease:1,
            reserved:1,
            fAckReq:1;
    short   cfFormat;
    BYTE    Value[1];
} DDEDATA;
```

The Value member should specify the location (drive and path) of the application's executable file, in the form of a null-terminated string of characters from the Windows character set. The character string has a maximum size of 128 characters (including the terminating null character). If the server sets the fAckReq bit, WinOldApp responds to the WM_DDE_DATA message by posting a WM_DDE_ACK message after processing the working-directory property. If the server sets the fRelease bit, WinOldApp frees the global memory object after copying the working-directory string. Otherwise, WinOldApp does not free the memory object.

Providing the Icon Property

If the shell is responding to WinOldApp's request for "GetIcon" property, the shell passes the following parameters with the WM_DDE_DATA message:

Parameter	Description
hwnd	Specifies the handle of WinOldApp's DDE window. The shell should use the handle that WinOldApp specified as the wParam parameter in the WM_DDE_REQUEST message.

message Specifies the WM_DDE_DATA message.

wParam Specifies the handle of the shell's DDE window.

LOWORD(lParam) Specifies a handle to a global memory object that contains a DDEDATA structure. A description of the contents of the DDEDATA structure follows this table. To report an error, the server should use one of the error values listed with the WinExec function.

HIWORD(lParam) Specifies an atom that represents the string, "GetIcon".

The low-order word of the lParam parameter is a handle to a global memory object that contains icon-property data. The icon data should be in the following form:

```
typedef struct tagICONPROPS { /* ip */

    unsigned reserved:12,   /* reserved                         */
             fResponse:1,   /* always 1                         */
             fRelease:1,    /* 1 if app. frees object, else 0   */
             reserved:1,    /* reserved                         */
             fAckReq:1;     /* 1 if app. should respond, else 0 */
    int      cfFormat;      /* clipboard format (not used)      */
    int      nWidth;        /* width, in pixels, of the icon    */
    int      nHeight;       /* height, in pixels, of the icon   */
    BYTE     nPlanes;       /* number of planes in XOR mask     */

    BYTE     nBitsPixel;    /* number of bits/pixel in XOR mask */
    LPBYTE   lpANDbits;     /* points to AND mask array         */
    LPBYTE   lpXORbits;     /* points to XOR mask array         */
} ICONPROPS;
```

If the server sets the fAckReq bit, WinOldApp responds to the WM_DDE_DATA message with a WM_DDE_ACK message after processing the data. If the server sets the fRelease bit, WinOldApp frees the global memory object after copying the working-directory string. Otherwise, WinOldApp does not free the memory object.

The lpANDbits and lpXORbits pointers may be either near or far. If the pointers are near (that is, the segment selector portion of the pointers is zero), the bits are part of the global memory object. The offset portion of the pointers is a near offset from byte zero of the object. Because the bits are part of the global memory object, they are freed along with the object. The combined size of the ICONPROPS structure together with the bits pointed to by the lpANDbits and lpXORbits members must be no more than 64K.

If the server needs to use far pointers for the lpANDbits and lpXORbits members, the bits must be part of a separate memory object. This object is not freed automatically when the global memory object is freed.

Terminating the DDE Conversation

The shell may terminate the conversation at any time by posting a WM_DDE_TERMINATE message. After WinOldApp has obtained its properties from the shell, it terminates the DDE conversation by posting a WM_DDE_TERMINATE message.

Additional reference words: 3.10 dosapp icon winexec
KBCategory:
KBSubcategory: UsrDdeOtherApps

PRB: Using HSZ in AFXEXT.H and DDEML.H
Article ID: Q98871

Summary:

SYMPTOMS

When using both the AFXEXT.H and DDEML.H include files in the same source file, the following message is received:

```
error C2371: 'HSZ' : redefinition; different basic types
```

CAUSE

The problem is a naming conflict between Dynamic Date Exchange Management Library (DDEML) and Visual Basic eXtension (VBX) files controls, in which both use the type HSZ [handle to a string that is zero (NULL) terminated] but with with different meanings.

AFXEXT.h includes the definition of HSZ in order to support the Control Development Kit (CDK), and DDEML.H uses HSZ as a string type.

RESOLUTION

Workarounds for this problem include:

- Separate the DDEML and VBX code into different files, thus preventing AFXEXT.H and DDEML.H from being included in the same source file.

-or-

- Use #define NO_VBX_SUPPORT in the file that implements DDE. This will cause DDEML's definition of HSZ to be used.

-or-

- Undefine one or the other HSZ in the .H files.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlRareMisc

INF: WM_DDE_EXECUTE Message Must Be Posted to a Window

Article ID: Q77842

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Chapter 15 of the "Microsoft Windows Software Development Kit Reference, Volume 2" documents the dynamic data exchange (DDE) protocol. The following statement is found on page 15-2:

> An application calls the SendMessage function to issue the WM_DDE_INITIATE message or a WM_DDE_ACK message sent in response to WM_DDE_INITIATE. All other messages are sent using the PostMessage function.

In the book "Windows 3: A Developer's Guide" by Jeffrey M. Richter (M & T Computer Books), the sample setup program uses the SendMessage function to send itself a WM_DDE_EXECUTE message that violates the DDE protocol and may not work in future versions of Windows.

More Information:

In Richter's sample, no real DDE conversation exists. The correct method to achieve the desired result is to use the SendMessage function to send a user-defined message to the window procedure. When this message is processed, proceed accordingly. For more information on user-defined messages, see chapter 6 of the "Microsoft Windows Software Development Kit Reference, Volume 1."

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDml

INF: Sample Code Demonstrates DDEML with Metafiles

Article ID: Q78807

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

DDEMETA is a file in the Software/Data Library that contains the source code to an application that demonstrates how to read a Windows metafile and pass the information using dynamic data exchange (DDE).

DDEMETA can be found in the Software/Data Library by searching on the word DDEMETA, the Q number of this article, or S13234. DDEMETA was archived using the PKware file-compression utility.

The remainder of this article explains how to use DDEMETA.

More Information:

DDEMETA is a server application that uses the Dynamic Data Exchange Management Library (DDEML).

To use DDEMETA, perform the following five steps:

1. Start DDEMETA.

2. Choose DDE Init from the Sample menu to register DDEMETA with the DDEML.

3. Start a DDE client application to initiate a conversation with DDEMETA. In the client, use the application name "DDEMETA" and the topic "Test".

4. After the conversation is established, the client application can request any "item" of type CF_METAFILEPICT from the server.

5. When DDEMETA receives the request, it will read the TEST.WMF file from the disk and pass it to the client application.

To test DDEMETA, request a CF_TEXT item. Note that the "item" reference is never checked; therefore, any item name works.

Note: This program will not read "placeable" metafiles. For more information on "placeable" metafiles, query on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and placeable

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

INF: Description of the DDEML Error Codes

Article ID: Q80082

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The documentation for version 3.1 of the Windows Software Development Kit (SDK) only states which of the Dynamic Data Exchange Management Library (DDEML) error values can be returned to each of the API functions. No descriptions of the different error values is included.

The following list describes each of the possible error codes that can be returned by the DDEML, its corresponding value, and a description of what may have caused the error.

Error	Error Value
DMLERR_ADVACKTIMEOUT	0x4000
DMLERR_DATAACKTIMEOUT	0x4002
DMLERR_EXECACKTIMEOUT	0x4005
DMLERR_POKEACKTIMEOUT	0x400B
DMLERR_UNADVACKTIMEOUT	0x4010

In each case, a synchronous transaction attempt has timed out.

DMLERR_BUSY 0x4001

The response to the transaction caused the DDE_FBUSY bit to be set.

DMLERR_DLL_NOT_INITIALIZED 0x4003

- A DDEML API was called without first calling the DdeInitialize function.
- An invalid IdInst parameter was passed to an API.

DMLERR_DLL_USAGE 0x4004

- An application initialized as a MONITOR has attempted DDE transactions.
- An application initialized as CLIENTONLY has attempted SERVER transactions.

DMLERR_INVALIDPARAMETER 0x4006

A parameter failed validation. Some of the possible causes are as

follows:

- The application used a data handle initialized with a different hszItem than that required by the transaction.

- The application used a data handle initialized with a different wFmt than that required by the transaction.

- The application used a client-side hConv with a server-side API or vice versa.

- The application used a freed data handle or hsz handle.

- More than one instance of the application used the same object.

DMLERR_LOW_MEMORY 0x4007

This error only happens in an XTYP_ERROR callback -- generally after a prolonged race condition (where the server application outruns the client) that consumes huge amounts of memory.

DMLERR_MEMORY_ERROR 0x4008

A memory allocation failed.

DMLERR_NOTPROCESSED 0x4009

A transaction failed -- generally with a NACK.

DMLERR_NO_CONV_ESTABLISHED 0x400A

A connection attempt failed to receive an ACK in reply.

DMLERR_POSTMSG_FAILED 0x400C

An internal PostMessage call failed.

DMLERR_REENTRANCY 0x400D

- A synchronous transaction was initiated while the application instance has another synchronous transaction in progress.
- The DdeEnableCallback function was called from within a callback.

DMLERR_SERVER_DIED 0x400E

- A server-side transaction has been attempted on a conversation that was terminated by the client.
- The server died mid-transaction.

DMLERR_SYS_ERROR 0x400F

A system API failed inside of the DDEML.

DMLERR_UNFOUND_QUEUE_ID 0x4011

An invalid transaction ID was passed to an API function. Once the application has returned from an XTYP_XACT_COMPLETE callback, the transaction ID for that callback is no longer valid.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

INF: DDEML CONVINFO Structure, wConvst Field Description

Article ID: Q81547

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The documentation for the Dynamic Data Exchange Management Library (DDEML) lists the possible values for the wConvst field, but it does not explain the causes for these values. The following table lists the possible values for the wConvst field and the causes of each:

wConvst	Cause
XST_ADVACKRCVD	The advise transaction was just completed.
XST_ADVDATAACKRCVD	The advise data transaction was just completed.
XST_ADVDATASENT	Advise data has been sent and is awaiting an acknowledgement.
XST_ADVSENT	An advise transaction is awaiting an acknowledgement.
XST_CONNECTED	The conversation has no active transactions.
XST_DATARCVD	The requested data has just been received.
XST_EXECACKRCVD	An execute transaction just completed.
XST_EXECSENT	An execute transaction is awaiting an acknowledgement.
XST_INCOMPLETE	The last transaction failed.
XST_INIT1	Mid-initiate state 1.
XST_INIT2	Mid-initiate state 2.
XST_NULL	Preinitiate state.
XST_POKEACKRCVD	A poke transaction was just completed.
XST_POKESENT	A poke transaction is awaiting an acknowledgement.
XST_REQSENT	A request transaction is awaiting an acknowledgement.
XST_UNADVACKRCVD	An unadvise transaction just completed.

```
XST_UNADVSENT       An unadvise transaction is awaiting an
                    acknowledgement.
```

Additional reference words: 3.10 DDEML
KBCategory:
KBSubcategory: UsrDml

INF: Sample Application Demonstrates Using DDEML

Article ID: Q82077

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

DMLDEMO is a file in the Software/Data Library that demonstrates how two applications for the Windows environment can communicate using the Dynamic Data Exchange Management Libraries (DDEML). DMLDEMO demonstrates the following types of transactions:

 - Connect
 - Request
 - Asynchronous Request
 - Hot Advise Loop
 - Warm Advise Loop

DMLDEMO can be found in the Software/Data Library by searching on the word DMLDEMO, the Q number of this article, or S13372. DMLDEMO was archived using the PKware file-compression utility.

Note: The DMLDEMO file contains code for a client application and for a server application in separate directories. To preserve the directory structure, be sure to specify the -d option switch to PKUNZIP. For example,

```
   pkunzip -d dmldemo
```


More Information:

The remainder of this article details the five transaction types.

Connect Transaction

The client calls the DdeConnect function to connect to the server and initiate a conversation. In the DMLDEMO application, the DdeConnect function starts a general conversation, which deals with text and bitmaps, and a SYSTEM conversation, which supports the system topic.

Request Transaction

When the client requires information from the server, the client calls the DdeClientTransaction function, specifying XTYP_REQUEST as the value for the uType parameter. The DMLDEMO client uses a request transaction to retrieve the number and type of bitmaps to display. The request transaction is used in both synchronous and asynchronous contexts.

Asynchronous Transaction

The client starts an asynchronous transaction with the server to request information that may not be ready at the time the client makes the request.

In the DMLDEMO application, the client starts an asynchronous transaction when the user chooses Asynchronous Transaction from the Transactions menu. When the server receives the request (as an XTYP_REQUEST transaction), it displays a dialog box that asks the user how many copies of the window bitmap the client should display. After the user dismisses the dialog box, the server returns a data handle containing the number the user selected from its callback function. When the server has completed the transaction, the DDEML sends an XTYP_XACT_COMPLETE to the client application's callback function. The client then displays the requested number of bitmaps.

In this case, an asynchronous transaction is required to prevent the transaction from timing out before the user has a chance to close the dialog box.

Hot Advise Loop

When a hot advise loop is established between a client and a server, the server notifies the client (through an XTYP_ADVDATA transaction) each time a certain piece of data changes. The notification includes a handle to the changed data. When a hot advise loop is active in the DMLDEMO application, the server notifies the client that the bitmap has changed (this notification includes a handle to the changed bitmap) every time the server's bitmap is changed. (To change the bitmap, choose Change Bitmap from the server application's menu.) The client application then displays the bitmap.

Warm Advise Loop

When a warm advise loop is established between a client and a server, the server notifies the client that the specified data has changed, but the server does not send the changed data to the client. In the DMLDEMO application, when the client receives a notification from the server, the client brings up a message box asking the user if the client's window should be updated with the new bitmap from the server. If the user chooses the "Yes" button, the client requests the changed bitmap from the server using the DdeClientTransaction function, specifying XTYP_REQUEST as the value of the uType parameter.

Additional reference words: 3.00 3.10 softlib DMLDEMO.ZIP
KBCategory:
KBSubcategory: UsrDml

INF: Freeing Memory in a DDEML Server Application

Article ID: Q83413

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

A Dynamic Data Exchange Management Library (DDEML) server application calls the DdeCreateDataHandle function to allocate a block of memory for data it will send to a client application. DdeCreateDataHandle returns a handle to a block of memory that can be passed between applications.

The server application owns every data handle it creates. However, it is not necessary to call DdeFreeDataHandle under every circumstance. This article details the circumstances under which the server application must call DdeFreeDataHandle and when the DDEML will automatically free a data handle.

More Information:

If the server application specifies the HDATA_APPOWNED flag in the afCmd parameter to DdeCreateDataHandle, it must explicitly call DdeFreeDataHandle to free the memory handle. Using HDATA_APPOWNED data handles is convenient when data, such as system topic information, is likely to be passed to a client application more than once, because the server calls DdeCreateDataHandle only once, regardless of the number of times the data handle is passed to a client application.

When it closes down, a server application must call DdeFreeDataHandle for each data handle that it has not passed to a client application. When the server creates a handle without specifying HDATA_APPOWNED, and passes the handle to a client application in an asynchronous transaction, the DDEML frees the data handle when the client returns from its callback function. Therefore, the server is not required to free the data handle it passes to a client because the DDEML frees the handle. However, if the data handle is never sent to a client application, the server must call DdeFreeDataHandle to free the handle. It is the client application's responsibility to call DdeFreeDataHandle for any data provided by a server in a synchronous transaction.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

INF: Executing Excel Functions with Return Values Using DDE
Article ID: Q83661

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the Windows environment, an application cannot use the WM_DDE_EXECUTE message to execute a Microsoft Excel function that returns a value. Excel enforces this limitation because it has no mechanism to provide the return value to the client application. If the client application is not interested in the function's return value, it can work around this limitation by posting a WM_DDE_EXECUTE message to Excel with commands to do the following:

 - Create a macro sheet.
 - Write the desired function call to the macro sheet.
 - Run the macro.

DDEXL is a sample in the Software/Date Library that demonstrates this technique. DDEXL allows the user to initiate a conversation with Excel, execute a series of Excel commands, which draw a chart (using CREATE.OBJECT), and terminate the conversation.

DDEXL can be found in the Software/Data Library by searching on the word DDEXL, the Q number of this article, or S13386. DDEXL was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib DDEXL.ZIP
KBCategory:
KBSubcategory: UsrDml

INF: Freeing Memory for Transactions in a DDEML Client App
Article ID: Q83912

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

A Dynamic Data Exchange Management Library (DDEML) client application can request data from a server both synchronously and asynchronously by calling the DdeClientTransaction function.

To make a synchronous request, the client application specifies XTYP_REQUEST as the value for the uType parameter to DdeClientTransaction, and any reasonable value for the uTimeout parameter.

To make an asynchronous request, the client application specifies XTYP_REQUEST as the value for the uType parameter to DdeClientTransaction, and TIMEOUT_ASYNC as the value for the uTimeout parameter.

The client can also establish an advise loop with a server application by specifying XTYP_ADVSTART as the value for the uType parameter. In an advise loop, the client application's callback function receives an XTYP_ADVDATA transaction each time the specified data item changes in the server application. (Note: This article discusses only hot advise loops in which changed data is communicated to the application. No data is transferred for a warm advise loop, only a notification that the data changed.)

The client application must free the data handle it receives from a synchronous transaction; however, the client application should not free the data handle it receives from an asynchronous transaction or from an advise loop.

More Information:

If the client application initiates a synchronous transaction, the DdeClientTransaction function returns a handle to the requested data. If the client application initiates an asynchronous transaction, the DdeClientTransaction function returns either TRUE or FALSE. When the data becomes available, the DDEML sends the client application an XTYP_XACT_COMPLETE notification accompanied by a handle to the requested data. In an active advise loop, the DDEML sends the client application an XTYP_ADVDATA notification accompanied by a handle to the updated data.

In the synchronous case, the client application must call DdeFreeDataHandle before it terminates to free a data handle (and the associated memory) that it received as the return value from DdeClientTransaction. If the DDEML server specified HDATA_APPOWNED when it created the data handle, then the data is invalidated when the

client calls DdeFreeDataHandle; the server must call DdeFreeDataHandle before terminating to free the associated memory.

In the asynchronous case, the DDEML sends the client application's callback function an XTYP_XACT_COMPLETE notification when the server has completed the transaction. A handle to the requested data accompanies the notification as the hData parameter to the callback function. This handle is valid until control returns from the client application's callback function. Once the client application's callback function returns control, the DDEML may free the data handle and the client application must not assume that the data handle received in the callback function remains valid. This fact has two implications, as follows:

- The client application cannot call DdeFreeDataHandle on the data handle it receives with an XTYP_XACT_COMPLETE transaction. If the client invalidates the data handle by freeing it in the client's callback function, and the DDEML later attempts to free the handle, a Fatal Exit will result.

- The client application must make a local copy of the data it receives with the XTYP_XACT_COMPLETE transaction to use that data after the callback function returns.

In an advise loop, the client application should not free the data handle that it receives as the hData parameter to the callback function. The DDEML frees the data handle when the client application returns from its callback function. If the client calls DdeFreeDataHandle on the data handle, the DDEML will cause a Fatal Exit when it attempts to free the same data handle.

These rules apply to all data handles, whether or not the server application specified the HDATA_APPOWNED flag when it created the handle.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

INF: CONVINFO Data Structure wStatus Field Description

Article ID: Q83916

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The documentation for the CONVINFO data structure used by the Dynamic Data Exchange Management Library (DDEML) lists the possible values of the structure's wStatus field. However, the documentation does not elaborate on the causes of these different values.

The following table lists the various values of the wStatus field and the conditions under which each status code is reported:

wStatus	Cause
ST_ADVISE	One or more links are in progress.
ST_BLOCKED	The conversation is blocked.
ST_BLOCKNEXT	The conversation will block after calling the next callback.
ST_CLIENT	The conversation handle passed to the DdeQueryConvInfo function is a client side handle. If the ST_CLIENT value is 0, then the conversation handle passed to the DdeQueryConvInfo function is a server side handle.
ST_CONNECTED	The conversation is connected.
ST_INLIST	The conversation is a member of a conversation list.
ST_ISLOCAL	Both sides of the conversation are using the DDEML.
ST_ISSELF	Both sides of the conversation are using the same instance of the DDEML.
ST_TERMINATED	The conversation has been terminated by the other side.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

PRB: GP Fault in DDEML from XTYP_EXECUTE Timeout Value
Article ID: Q83999

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

SYMPTOMS

When the following occurs

1. A DDEML (Dynamic Data Exchange Server Library) server application requires more time to process a XTYP_EXECUTE transaction than the timeout value specified by a DDEML client application

2. The server application creates windows as part of its processing

3. The client application abandons the transaction because the transaction timed out

a GP fault occurs in the DDEML.

CAUSE

The server application receives a window handle with the same value as the hidden window created to control the transaction.

RESOLUTION

Specify a timeout value in the client application longer than the time required by the server application to complete the task.

More Information:

To use DDEML, an application (either a client or a server) registers a callback function with the library. The DDEML calls the callback function for any DDE activity. A DDE transaction is similar to a message; it contains a named constant, accompanied by other parameters.

A client application issues a XTYP_EXECUTE transaction to instruct the server application to execute a command. When a client calls the DdeClientTransaction function to issue a transaction, it can specify a timeout value, which is the amount of time (in seconds) the client is willing to wait while the server processes the transaction. If the server fails to execute the command within the specified timeout value, the DDEML sends a message to the client that the transaction timed out. Upon receipt of this message, the client can inform the user, reissue the command, abandon the transaction, or take other appropriate actions.

If a client application specifies a short timeout period (one second, for example) and the server requires fifteen seconds to execute a command, the client will receive notification that the transaction

timed out. If the client terminates the transaction, which is an appropriate action, the DDEML will GP fault.

When the client sends an XTYP_EXECUTE transaction, the DDEML creates a hidden window for the conversation. If the client calls the DdeAbandonTransaction function to terminate the transaction, the DDEML destroys the associated hidden window.

At the same time, the server application processes the execute transaction, which might involve creating one or more windows. If the server creates a window immediately after the DDEML destroys a window, the server receives a window handle with the same value as that of the destroyed window. After the server completes processing the execute transaction, it returns control to the DDEML.

Normally, the DDEML determines that the callback function is returning to a conversation that has been terminated. It calls the IsWindow function with the window handle for the transaction's hidden window to ensure that the handle remains valid.

Because the window handle has been allocated to the server application, the IsWindow test succeeds. However, this handle no longer corresponds to the transaction's hidden window. Therefore, when the DDEML attempts to retrieve the pointer kept in the hidden window's window extra bytes, the pointer is not available. When the DDEML uses the contents of this memory, a GP fault is likely to result.

The current way to work around this problem is to specify a timeout value in the client application that is longer than the time required by the server to complete its processing.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

INF: Application Can Allocate Memory with DdeCreateDataHandle

Article ID: Q85680

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

An application can use the DdeCreateDataHandle function to create a handle to a block of data. The application can use the handle to pass the data to another application in a dynamic data exchange (DDE) conversation using the Dynamic Data Exchange Management Libraries (DDEML). All DDEML functions refer to blocks of memory using data handles.

An application can allocate memory and manually create a data handle associated with the memory (using method 1 below), or automatically by using the DdeCreateDataHandle function (method 2 below).

Method 1

1. Obtain a block of memory using the GlobalAlloc or LocalAlloc function or by declaring a variable in your application.

2. Fill the block of memory with the desired data.

3. Call the DdeCreateDataHandle function to create a data handle associated with the block of memory.

Method 2

1. Call the DdeCreateDataHandle function with the lpvSrcBuf parameter set to NULL, the cbInitData parameter set to zero, and the offSrcBuf parameter set to the number of bytes of memory required.

2. To retrieve a handle to the memory block, specify the data handle returned by DdeCreateDataHandle as the hData parameter of the DdeAccessData function. This operation is similar to calling the GlobalLock function on a handle returned from GlobalAlloc.

3. Use the pointer to fill the memory block with data.

4. Call DdeUnaccessData to unaccess the object. This operation is similar to calling the GlobalUnlock function on a handle returned from GlobalAlloc.

The following code fragment demonstrates method 2:

```
   // Retrieve the length of the data to be stored
   cbLen = lstrlen("This is a test") + 1;
```

```
   // Create the data handle and allocate the memory
   hData = DdeCreateDataHandle(idInst, NULL, 0, cbLen,
                               hszItem, wFmt, 0);

   // Access the data handle
   lpstrData = (LPSTR)DdeAccessData(hData, NULL);

   // Fill the block of memory
   lstrcpy(lpstrData, "This is a test");

   // Unaccess the data handle
   DdeUnaccessData(hData);
```

When an application obtains a data handle from DdeCreateDataHandle, the application should next call DdeAccessData with the handle. If a data handle is first specified as a parameter to a DDEML function other than DdeAccessData, when the application later calls DdeAccessData, the application receives only read access to the associated memory block.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDml

PRB: DdeUnaccessData Function Documented Incorrectly
Article ID: Q86006

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The DdeUnaccessData function is incorrectly documented on page 196 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" manual, as follows:

   The DdeUnaccessData function frees a global memory object.

The DdeUnaccessData function "unaccesses" a global memory object accessed by DdeAccessData. These functions are similar to GlobalUnlock and GlobalLock, respectively.

Additional reference words: 3.10 docerr
KBCategory:
KBSubcategory: UsrDml

INF: Software Library Has DDE Management Library Information

Article ID: Q89542

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

BOUNCET is a file in the Software/Data Library that details how to use the Dynamic Data Exchange Management Library (DDEML) to perform dynamic data exchange (DDE) in an application developed for the Microsoft Windows environment.

Another file in the Software/Data Library, BOUNCE, contains sample source code that implements a DDE client application and a DDE server application using the services of the DDEML.

BOUNCET can be found in the Software/Data Library by searching on the keyword BOUNCET, the Q number of this article, or S13593. BOUNCET was archived using the PKware file-compression utility.

BOUNCE filename can be found in the Software/Data Library by searching on the keyword BOUNCE, the Q number of this article, or S13560. BOUNCE was archived using the PKware file-compression utility.

Additional reference words: 3.00 3.10 softlib BOUNCE.ZIP BOUNCET.ZIP
KBCategory:
KBSubcategory: UsrDml

INF: Do Not Forward DDEML Messages from a Hook Procedure

Article ID: Q89828

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

If an appication for Windows uses the Dynamic Data Exchange Management Library (DDEML) in addition to a message hook [for example, by calling SetWindowsHook() or SetWindowsHookEx()], it is possible that your hook procedure will receive messages that are intended for the DDEML libraries.

For the DDEML libraries to work properly, you must make sure that your hook function does not forward on any messages that are intended for the DDEML libraries.

More Information:

If your hook procedure receives a code of type MSGF_DDEMGR, you should return FALSE instead of calling the CallNextHookEx() function.

The way to handle this situation is to use the following code:

```
   if (MSGF_DDEMGR == code)
      return FALSE;
   else
   {
   ...
   }
```

For more information about message hooks and DDEML, please see the above mentioned functions in the manual or the online help facility.

Additional reference words: 3.0 3.00 3.1 3.10
KBCategory:
KBSubcategory: UsrDml

INF: Sample: DDEML Samples Using Microsoft Foundation Classes

Article ID: Q92829

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

CDDEML is a set of two sample applications that demonstrate a DDEML client and a DDEML server. They are written using the Microsoft Foundation Class Libraries (MFC) and define new classes for handling DDEML.

CDDEML is available in the Software/Data Library and can be found by searching on the word CDDEML, the Q number of this article, or S13692. CDDEML was archived using PKware file-compression utility.

More Information:

This sample contains two applications that demonstrate a DDEML server and DDEML client in MFC applications. This sample defines four new classes to help implement the DDEML support. These classes are:

    CDDEClient
    CDDEClientConv
    CDDEServer
    CDDEServerConv

The CDDEClient and CDDEServer classes encapsulate the functionality of a DDEML client or server. CDDEClientConv and CDDEServerConv encapsulate individual conversations that are managed by the CDDEClient or CDDEServer class. The CDDEClient and CDDEServer classes maintain a map of the current conversations according to the conversations handle. Using this map, the server or client can look up any ongoing conversation based on its handle.

These classes were designed to implement DDEML in a very general way. As a result, these classes don't do much by themselves; they are designed to be reused in an application by deriving from each of these classes. Typically, an application derives one class from CDDEClient or CDDEServer. This derived class contains details such as what conversations are supported and how to handle callbacks that do not relate to a specific conversation. Usually there is only one instance of this class in an application. An application may derive many classes from CDDEClientConv or the application supports. These derived classes implement how each conversion handles callbacks. They also may contain buffers to store data from the conversation, or pointers to other objects that they need to communicate with. This sample contains a client and server application built using the classes mentioned above.

One difficulty in writing DDEML classes is how to handle the DDEML callback function. In this sample, the DDEML callback function is contained in the CDDEClient or CDDEServer class. The callback function first checks to see

whether the transaction is for the client or server, or for one of the conversations. If it is for the client or server, then the callback function calls the member function that handles that transaction. If the transaction is for a conversation, then the callback function looks up the pointer to the conversation object using the handle of the conversation in the conversation map. Then it calls the member function of the conversation that handles that transaction. Therefore, by using these classes, an application developer does not need to write a DDEML callback function.

Additional reference words: 3.00 3.0 3.10 3.1
KBCategory:
KBSubcategory: UsrDmlRare/misc

SAMPLE: Shell DDE Using DDEML

Article ID: Q99807

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

The Windows Program Manager supports a dynamic data exchange (DDE) command-string interface that allows other applications to create, display, delete, and reload groups; add, replace, and delete items from groups; and to close Program Manager.

PMDDEML is a file in the Software/Data library that demonstrates how to use this Program Manager DDE command-string interface using the Dynamic Data Exchange Management Library (DDEML) introduced in Microsoft Windows version 3.1.

PMDDEML can be found in the Software/Data Library by searching on the word PMDDEML, the Q number of this article, or S14197. PMDDEML was archived using the PKware file-compression utility.

MORE INFORMATION
================

PMDDEML demonstrates only a small subset of the command-strings supported by Program Manager. The commands demonstrated by PMDDEML include:

CreateGroup
ShowGroup (both minimize and restore)
DeleteGroup

The other command-strings supported by Program Manager can be easily implemented in a manner similar to those commands demonstrated by PMDDEML.

The DDEML functions demonstrated by PMDDEML include:

DdeInitialize
DdeUninitialize
DdeConnect
DdeDisconnect
DdeCreateStringHandle
DdeFreeStringHandle
DdeClientTransaction

To use PMDDEML, run it and select the Shell Commands menu. Choose the Create "Sample Group" menu item. This will create a new program group (if one does not already exist) in Program Manager. Once the sample group is created, the group can be minimized, restored, or deleted by

selecting the appropriate menu item in PMDDEML.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlProgman

PRB: DDESPY GP Faults Upon Return of CBR_BLOCK
Article ID: Q102549

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

SYMPTOMS
========

DDESPY GP faults when a DDEML server application's dynamic data exchange (DDE) callback function returns CBR_BLOCK from a transaction of XCLASS_DATA class.

CAUSE
=====

Expecting to always obtain a valid data handle from transactions of XCLASS_DATA class, DDESPY calls DdeAccessData() on the hData it receives in an attempt to dump the data to its output window. Internally, DDEML's DdeAccessData() translates to a GlobalLock() call. Consequently, a return value of CBR_BLOCK (defined in WINDOWS.H as -1) would mean calling GlobalLock() on an invalid hData == -1, and thus results in a general protection (GP) fault.

STATUS
======

Microsoft has confirmed this to be a problem in the version of DDESPY that comes with the Windows 3.1 SDK. With the improved parameter validation in Windows NT, this problem should be corrected in the Windows NT version.
MORE INFORMATION
================

In DDEML, transactions of XCLASS_DATA class are typically expected to return a data handle. (Among the DDEML transactions of this class are XTYP_REQUEST, XTYP_ADVREQ, and XTYP_WILDCONNECT.) However, in situations where such transactions require lengthy processing (such as a server gathering data from a network), thereby making it impossible to return a data handle immediately, an application may choose to return CBR_BLOCK from its DDE callback function. This allows the server application to gather data "asynchronously," as DDEML queues up further transactions on the current conversation.

Using DDEPSY to monitor this type of activity in the system results in a GP fault, because DDESPY calls DdeAccessData() on an invalid hData (== -1), as described in the CAUSE section above.

More information on DDEML transaction classes can be found in Section 5.8.7 of the Windows 3.1 SDK "Programmer's Reference, Volume 1: Overview." More information on returning CBR_BLOCK to suspend DDEML transactions may be found in Section 5.8.6 of the Windows 3.1 SDK

"Programmer's Reference, Volume 1: Overview," or by searching on the
following words in the Microsoft Knowledge Base:

   DDEML and CBR_BLOCK

Additional reference words: 3.10 GPF gp-fault
KBCategory:
KBSubcategory: UsrDmlBlocking

INF: DdeCreateStringHandle() lpszString param Docerr

Article ID: Q102570

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY

The documentation for the DdeCreateStringHandle() function in the Windows 3.1 Software Development Kit (SDK) "Programmer's Reference, Volume 2: Functions" incorrectly states that the lpszString parameter may point to a buffer of a null-terminated string of any length, when the string is actually limited to 255 characters.

DDEML string-management functions are internally implemented using global atom functions. DdeCreateStringHandle() in particular, internally calls GlobalAddAtom(), and therefore inherits the same limitation as atoms to a maximum of 255 characters in length.

MORE INFORMATION

DDEML applications use string handles extensively to carry out DDE tasks. To obtain a string handle for a string, an application calls DdeCreateStringHandle(). This function registers the string with the system by adding the string to the global atom table, and returns a unique value identifying the string.

The global atom table table in Windows can maintain strings that are less than or equal to 255 characters in length. Any attempt to add a string of greater length to this global atom table will fail. Hence, a call to DdeCreateStringHandle() fails for strings over 255 characters long.

This limitation is by design. DDEML applications that use the DdeCreateStringHandle() function should conform to the 255-character limit.
This limitation has been preserved on the Windows NT version of DDEML for compatibility reasons.

Additional reference words: 3.10 documentation error
KBCategory:
KBSubcategory: UsrDmlStringhan

INF: Calling DdePostAdvise() from XTYP_ADVREQ
Article ID: Q102571

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

The documentation for DdePostAdvise() in the Windows 3.1 Software Development Kit "Programmer's Reference, Volume 2: Functions" states the following in the Comments section:

   If a server calls DdePostAdvise() with a topic/item/format name set that includes the set currently being handled in an XTYP_ADVREQ callback, a stack overflow may result.

MORE INFORMATION
================

This is merely a warning against calling DdePostAdvise() from within a DDE callback function's XTYP_ADVREQ transaction, because it may result in a stack overflow.

Like window procedures, DDE callbacks must be coded with care to avoid infinite recursion (eventually resulting in a stack overflow). Because DdePostAdvise() causes DDEML to send an XTYP_ADVREQ transaction to the calling application's DDE callback function, calling DdePostAdvise() on the same topic/item/format name set as the one currently being handled results in an infinite loop.

An analogous piece of code that has become a classic problem in Windows programming involves calling UpdateWindow() in a WM_PAINT case:

```
   case WM_PAINT:
       InvalidateRect (hWnd, NULL, TRUE);
       UpdateWindow (hWnd);
```

Calling UpdateWindow() as in the code above causes a WM_PAINT message to be sent to a window procedure, and thus results in the same type of infinite recursion that occurs when calling DdePostAdvise() from an XTYP_ADVREQ transaction.

An example of a situation that would lend itself to this scenario would be one where data needs to be updated as a result of a previous data change. There are two ways to work around the stack overflow problem in this case:

 - Post a user-defined message and handle the data change asynchronously. For example,

```
      // in DdeCallback:
```

```
      case XTYP_ADVREQ:
               if ((!DdeCmpStringHandles (hsz1, ghszTopic)) &&
                   (!DdeCmpStringHandles (hsz2, ghszItem)) &&
                   (fmt == CF_SOMEFORMAT))
                 {
                       HDDEDATA  hData;

                       hData = DdeCreateDataHandle ();
                       PostMessage (hWnd, WM_DATACHANGED,hData,);
                       return (hData);
                 }
                 break;

       // in MainWndProc():
       case WM_DATACHANGED:
               DdePostAdvise (idInst, ghszTopic, ghszItem);
                 :
```

- Return CBR_BLOCK from the XTYP_ADVREQ and let DDEML suspend further transactions on that conversation, while the server prepares data asynchronously.

More information on how returning CBR_BLOCK allows an application to process data "asynchronously" may be derived from Section 5.8.6 of the Windows 3.1 Software Development Kit (SDK) "Programmer's Reference, Volume 1: Overview," or by querying on the following words in the Microsoft Knowledge Base:

DDEML and CBR_BLOCK

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlRaremisc

INF: Changes Between Win 3.1 and WFW 3.1 Versions of DDEML

Article ID: Q102572

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

The following changes have been made from the Windows 3.1 version of DDEML to the version that shipped with Windows for Workgroups (WFW) version 3.1:

- The default shutdown timeout was changed from 3,000 milliseconds to 30,000 milliseconds.

- DdeQueryConvInfo() added support for the new hwnd and hwndPartner fields.

- DdeQueryConvInfo() added more robust handling for improper setting of the CONVINFO structure's cb field when called.

- Rewrote DdePostAdvise() to supply the correct count of remaining transactions to be processed on the same topic!item!format name set for the XTYP_ADVREQ transaction.

- Disallow synchronous calls if the application is shutting down, or if already in the middle of a synchronous transaction.

- DdeUninitialize() was fixed to defer processing if called while in the middle of a synchronous transaction.

- DdeUninitialize() was fixed to dispatch any pending DDE messages and to make necessary callbacks before quitting.

- Added call to DdeUninitialize() from within DdeClientTransaction() when in the middle of a synchronous transaction and DdeUninitialize() had been called.

- Allow any instance to call DdeCreateStringHandle().

- Have DdeAbandonTransaction() only mark abandoned transactions, rather than trying to remove the transaction from internal structures.

- Alter PostMessage() overflow queuing to queue based on target instead of on the same task, so that one target queue overflow does not bog down all PostMessage() calls.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlRaremisc

INF: XTYP_EXECUTE and its Return Value Limitations
Article ID: Q102574

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

A DDEML client application can use the XTYP_EXECUTE transaction to cause a server to execute a command or a series of commands. To do this, the client creates a buffer that contains the command string, and passes either a pointer to this buffer or a data handle identifying the buffer, to the DdeClientTransaction() call.

If the server application generates data as a result of executing the command it received from the client, the return value from the DdeClientTransaction() call does not provide a means for the client to access this data.

MORE INFORMATION
================

For an XTYP_EXECUTE transaction, the DdeClientTransaction() function returns TRUE to indicate success, or FALSE to indicate failure. In most cases, this provides inadequate information to the client regarding the actual result of the XTYP_EXECUTE command.

Likewise, the functionality that DDEML was supposed to provide through the lpuResult parameter of the DdeClientTransaction() function upon return is currently not supported, and may not be supported in future versions of DDEML. The lpuResult parameter was initially designed to provide the client application access to the server's actual return value (for example, DDE_FACK if it processed the execute, DDE_FBUSY if it was too busy to process the execute, or DDE_FNOTPROCESSED if it denied the execute).

In cases where the server application generates data as a result of an execute command, the client has no means to get to that data, nor does it have a means to determine the status of that execute command through the DdeClientTransaction() call.

An example of this might be one where the DDEML client application specifies a command to a server application such as "OpenFile <FileName>" to open a file, or "DIR C:\WINDOWS" to get a list of files in a given directory.

There are two ways that the client application can work around this limitation and gain access to the data generated from the XTYP_EXECUTE command:

Method 1

The client can issue an XTYP_REQUEST transaction (with the item name set to "ExecuteResult", for example) immediately after its XTYP_EXECUTE transaction call returns successfully. The server can then return a data handle in response to this request, out of the data generated from executing the command.

Method 2

The client can establish an ADVISE loop with the server (with topic!item name appropriately set to Execute!Result, for example) just before issuing the XTYP_EXECUTE transaction. As soon as the server then executes the command, it can immediately update the advise link by calling DdePostAdvise(), and return a data handle out of the data generated from executing the command. The client then receives the data handle in its callback function, as an XTYP_ADVDATA transaction.

Note that these workarounds apply only if one has access to the server application's code. Third-party server applications that provide no means to modify their code as described above can't obtain any data generated by the application as a result of an XTYP_EXECUTE back to the client.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlExecute

INF: Obtaining Group/Item Info from ProgMan Using DDEML
Article ID: Q102575

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

SUMMARY
=======

Dynamic-data exchange (DDE) can be used to obtain group and/or item information from Program Manager. Many applications, especially setup applications, need this information before items or groups are added/deleted from Program Manager. Information such as group names, group filenames, the number of items in a group, an item's working directory, title, xpos, ypos, and so forth, can be obtained very easily from Program Manager via DDE. This article discusses how to obtain this type of information from Program Manager using DDEML.

MORE INFORMATION
================

Program Manager is a DDEML server that provides DDE client applications with valuable information regarding group windows and items inside the group windows.

DDEML client applications have to connect to Program Manager with PROGMAN/PROGMAN as SERVICE/TOPIC names. (Program Manager also supports SHELL/APPPROPERTIES service/topic names.) Once connected, the client application needs to REQUEST group/item information from Program Manager by issuing a request transaction.

Group Information

To obtain names of all the groups currently in Program Manager, the client has to issue a request transaction with the item name set to "GROUPS". The request has to be made in CF_TEXT format.

The return value from this transaction is a data handle that contains the names of all the groups in Program Manager. Each group name is separated by a carriage return (\r\n). Below is sample code that requests names of all the groups from Program Manager:

```
        ghszItem = DdeCreateStringHandle(idInst,"GROUPS",
 CP_WINANSI);
        hProgData = DdeClientTransaction(NULL,
// pointer to data to pass
                                                   // to server
                        0,            // length of data
                        ghszConv,     // handle to conversation
                        ghszItem,     // handle to item-name
                                                   // string
```

```
                     CF_TEXT,      // clipboard format
                     XTYP_REQUEST, // transaction type
                     5000,         // transaction timeout
                     NULL);       // pointer to result


     retVal =  DdeGetData(hProgData, (void FAR*)szBuffer,
                              sizeof(szBuffer),  0) ;
        MessageBox(NULL, szBuffer, "TEST", MB_OK) ;
```

If there are four groups in Program Manager (for example, Main, Accessories, SDK, and Applications), then the message-box call will display names of all these groups, one per line.

Item Information

To obtain information regarding all the items in a group, the client application has to issue a request transaction to Program Manager, this time with the item name set to the name of the group. For example, if the client application needs to find out if an item exists in Program Manager group "Main", or if it needs to find out all the items that are in group Main, then the call to DdeCreateStringHandle() above must be changed to

```
   ghszItem = DdeCreateStringHandle(idInst,"MAIN", CP_WINANSI);
```

to create a string handle for the appropriate item name, Main.

The return value from this transaction is a data handle that contains names of all the items in the group Main. A typical example of the data inside resembles the following:

```
   "MAIN",C:\WINDOWS\MAIN.GRP,5,3
   "DOS","DOSPRMPT.PIF",,PROGMAN.EXE,166,4,9,0,0
   "Mail","MSMAIL.EXE",,C:\WINDOWS\MSMAIL.EXE,258,2,0,0,0
   "Write","WRITE.EXE",,C:\WINDOWS\WRITE.EXE,384,48,0,0,0
   "File","WINFILE.EXE",,C:\WINDOWS\WINFILE.EXE,6,2,0,0,0
   "Brief ","B.PIF",,PROGMAN.EXE,93,49,2,0,0
```

Program Manager provides the list in CF_TEXT format. The fields of group information are separated by commas. The first line of the information contains the group name (in quotation marks), the path of the group file, and the number of items in the group.

NOTE: The last value on line 1 is not documented in the Windows Software Development Kit (SDK); it can be any value between 1 and 8. This is the same value that is passed as the second param to the ShowGroup() command. This value indicates the state of the group window, whether minimized, active, and so forth.

In this particular case, the group window Main is currently the active group and is maximized.

For more information on this value, please refer to the ShowGroup() command documentation in Section 17.2.2 of the Windows 3.1 SDK "Programmer's Reference, Volume 1: Overview."

Each subsequent line contains information about an item in the group, including the command line (in quotation marks), the default directory, the icon path, the position in the group, the icon index, the shortcut key (in numeric form), and the minimize flag.

This set of information for each item corresponds to the parameters passed to the AddItem() function when each item was added to the group. For more information on each of these parameters, refer to the AddItem() command documentation on Section 17.2.5 of the Windows 3.1 SDK, "Programmer's Reference, Volume 1: Overview."
Additional reference words: 3.10 GPF gp-fault progman
KBCategory:
KBSubcategory: USRDmlProgman

PRB: DDEML Fails to Call TranslateMessage() in its Modal Loop
Article ID: Q102576

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

During a synchronous transaction, DDEML causes the client to enter a modal loop until the transaction is processed. While DDEML dispatches messages appropriately, it fails to call TranslateMessage() while inside this modal loop. This problem does not apply to asynchronous transactions, where no such modal loop is entered.

SYMPTOMS
========

A common symptom of this problem is seen as the client processes user input while inside DDEML's modal loop in a synchronous transaction. WM_KEYDOWN and WM_KEYUP messages are received, with no corresponding WM_CHAR message for the typed character.

CAUSE
=====

No WM_CHAR message is received because the WM_KEYDOWN message is never translated. For this to take place, a call to TranslateMessage() must be made inside the modal loop.

RESOLUTION
==========

This limitation is by design. DDEML applications can work around this limitation by installing a WH_MSGFILTER hook, watching out for code == MSGF_DDEMGR.

The WH_MSGFILTER hook allows an application to filter messages while the system enters a modal loop, such as when a modal dialog box (code == MSGF_DIALOGBOX) or a menu (code == MSGF_MENU) is displayed; and similarly, when DDEML enters a modal loop in a synchronous transaction (code == MSGF_DDEMGR).

The Windows 3.1 Software Development Kit (SDK) DDEML\CLIENT sample demonstrates how to do this in DDEML.C's MyMsgFilterProc() function:

```
/***********************************************************************
 *
 *  FUNCTION: MyMsgFilterProc
 *
 *  PURPOSE:  This filter proc gets called for each message we handle.
 *            This allows our application to properly dispatch messages
 *            that we might not otherwise see because of DDEMLs modal
```

```
 *            loop that is used while processing synchronous transactions.
 *
***********************************************************************/

   DWORD FAR PASCAL MyMsgFilterProc( int nCode, WORD wParam,

                DWORD lParam)
   {
       wParam; // not used

    #define lpmsg ((LPMSG)lParam)

    if (nCode == MSGF_DDEMGR) {

           /* If a keyboard message is for the MDI, let the MDI client
            * take care of it. Otherwise, check to see if it's a normal
            * accelerator key. Otherwise, just handle the message as usual.
            */

           if ( !TranslateMDISysAccel (hwndMDIClient, lpmsg) &&
                !TranslateAccelerator (hwndFrame, hAccel, lpmsg)){
               TranslateMessage (lpmsg);
               DispatchMessage (lpmsg);
           }
           return(1);
       }
       return(0);
   #undef lpmsg
   }
```

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlRaremisc

INF: Returning CBR_BLOCK from DDEML Transactions
Article ID: Q102584

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

DDEML servers are applications that provide data to client applications. For some servers, this data gathering may be a lengthy process, as when gathering data from sources such as serial ports or a network. DDEML allows a server application to process data asynchronously in these situations by returning CBR_BLOCK from the DDE callback function.

MORE INFORMATION
================

In DDEML-based applications, while transactions can be either synchronous or asynchronous, only DDEML client applications may choose to establish either type of transaction when requesting data from a server application. DDEML server applications do not distinguish between synchronous and asynchronous transactions.

Asynchronous transactions can be very useful when client applications know that the partner server application will take some time to gather data. This type of transaction frees up the client to do other things while waiting for a notification from the server of data availability.

Server applications have no way of determining whether the client application has requested data synchronously or asynchronously. Request transactions on the server's side are always synchronous. When a client requests data, the server's callback receives an XTYP_REQUEST transaction, where the expected return value is a data handle. If the server application has to wait for data from a serial port, for example, access to the CPU by other applications will be delayed, thereby freezing the system until data arrives.

There are a couple of ways one can enable the server to gather data in an asynchronous manner, thereby allowing it to yield to other applications on the system while it gathers data. One method is to use CBR_BLOCK; another is to change the request transaction to a one-time ADVISE loop.

Method 1

Given that DDEML callbacks are not re-entrant, and that DDEML expects a data handle as a return value from the XTYP_REQUEST transaction (and transactions of XCLASS_DATA class), the server application can block the callback momentarily. It can do this by returning a CBR_BLOCK

value after posting itself a user-defined message.

This way, the server application can gather data in the background while DDEML queues up any further transactions. The server can start gathering data when its window procedure gets the user defined message that was posted by its DDE callback function.

When a server application returns CBR_BLOCK for a request transaction, DDEML disables the server's callback function. It also queues transactions that are sent by DDEML after its callback has been disabled. This feature gives the server an opportunity to gather data while allowing other applications to run in the system.

As soon as data becomes available, then the server application can call DdeEnableCallBack() to re-enable the server callback function. Once the callback is re-enabled, DDEML will resend the same request transaction to the server's callback and this time, because data is ready, the server application can return the appropriate data handle to the client.

Transactions that were queued up because of an earlier block are sent to the server's callback function in the order they were received by DDEML.

The pseudo code to implement method 1 might resemble the following:

```
BOOL gbGatheringData = TRUE;   // Defined GLOBAlly.
HDDEDATA ghData = NULL;

HDDEDATA CALLBACK DdeServerCallBack(...)
   {
       switch(txnType)
       {
       case XTYP_REQUEST:

          // If the server takes a long time to gather data...
          // for this topic/item pair, then
          // post a user-defined message to the server app's wndproc
       // and return CBR_BLOCK... DDEML will block the callback
       // and queue transactions.

       if(bGatheringData) {
         PostMessage(hSrvWnd, WM_GATHERDATA, .....) ;
         return CBR_BLOCK;
       }
       else              // Data is ready, send back handle.
         return ghData;

       default:
         return DDE_FNOTPROCESSED;
      }
    }

LRESULT CALLBACK SrvWndProc(...)
  {
    switch (wMessage)
       {
```

```
          case WM_GATHERDATA:

             while (bGatheringData)
               {
                  // Gather data here while yielding to others
                  // at the same time!
      if(!PeekMessage(..))
        bGatheringData = GoGetDataFromSource (&ghData);
                  else {
                    TranslateMessage() ;
                    DispatchMessage ();
                  }
                }
          DdeEnableCallback (idInst, ghConv, EC_ENABLEALL);
             break ;

          default:
            return DefWndProc();
        }
    }
```

Method 2

Advise transactions in DDEML (or DDE) are just a continuous request link. Changing the transaction from a REQUEST to a "one time only" ADVISE loop on the client side allows the server to gather data asynchronously.

The client application can start an ADVISE transaction from its side and when the server receives a XTYP_ADVSTART transaction, return TRUE so that an ADVISE link is established. Once the link is established, the server can start gathering data, and as soon as it becomes available, notify the client of its availability.

This can be done by calling DdePostAdvise(). The server can use PeekMessage() to gather data if the data gathering process is a lengthy one, so that other applications on the system will get a chance to run. Once the client receives the data from the server in its callback (in its XTYP_ADVDATA transaction), it can disconnect the the ADVISE link from the server by specifying an XTYP_ADVSTOP transaction.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDmlBlocking

INF: Using Main Window Edit Menu with Dialog Box Edit Controls

Article ID: Q38170

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When a dialog box has an edit control, it may be helpful if the commands from the Edit menu on the application's main window can be used to change the text. The following factors must be considered when this is done:

- Where does the input focus go when the editing operation is complete?

- What happens if no text is selected when cut, copy, or clear is chosen?

- If more than one edit control is visible, on which control should the action be taken?

When the main window receives a command from its Edit menu, the corresponding command is sent to the appropriate edit control.

CREATDLG is a file in the Software/Data Library that demonstrates this procedure. The code is commented, and a README.TXT file that discusses coding issues is included.

CREATDLG can be found in the Software/Data Library by searching on the word CREATDLG, the Q number of this article, or S12127. CREATDLG was archived using the PKware file-compression utility.

Additional reference words: 2.00 2.03 2.10 3.00 2.x SR# G881027-5331
KBCategory:
KBSubcategory: UsrDlgsKeysinput

INF: Modal Dialog Child of Modeless Dialog Box Sample Code
Article ID: Q77783

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

MDLMDLS is a file in the Software/Data Library that demonstrates how to create a modal child of a modeless dialog box. The sample uses two modeless dialog boxes. However, when the second dialog is created, the first dialog and the main application window are disabled.

MDLMDLS can be found in the Software/Data Library by searching on the word MDLMDLS, the Q number of this article, or S12159. MDLMDLS was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib MDLMDLS.ZIP
KBCategory:
KBSubcategory: UsrDlgsModal

INF: ENTER & TAB Keys in a Dialog Box Multiline Edit Control

Article ID: Q76474

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In an application running under Windows version 3.0 or 3.1, when the user interacts with a dialog box and presses the ENTER key, the application accepts any changes made to information in any control of the dialog box and dismisses the dialog box. However, when the dialog box contains a multiline edit control (MLE), it may be considered more intuitive for the user to use the ENTER key to advance to the next line in the edit control. Windows 3.0 and 3.1 use the CTRL+ENTER key combination for this purpose.

Similarly, Windows uses the TAB key to move the input focus to the next control in the tabbing sequence in a dialog box. However, in an edit control, it might be useful for the user to be able to enter TAB characters. Windows uses the CTRL+TAB key combination for this purpose.

Windows version 3.1 defines the ES_WANTRETURN edit control style, which changes the behavior of the ENTER key in an MLE in a dialog box. If an application is designed to run only under Windows versions 3.1 and later, specify the ES_WANTRETURN edit control style. None of the three techniques described below is necessary.

This article discusses three different methods that can be used to modify the behavior of the ENTER key in an MLE in an application's dialog box. These methods are compatible with Windows versions 3.0 and 3.1. Methods 1 and 3 below can be adapted to modify the behavior of the TAB key, if desired.

A major disadvantage of any of these methods is that the ENTER or TAB key will no longer act as it does in other applications, and its behavior no longer will be what an experienced Windows user would expect.

MLEENTER is a file in the Software/Data Library that demonstrates two of the methods described in this document. MLEENTER can be found in the Software/Data Library by searching on the word MLEENTER, the Q number of this article, or S13007. MLEENTER was archived using the PKware file-compression utility.

More Information:

Under versions of Windows earlier than 3.0, an edit control subclass procedure could respond to the WM_GETDLGCODE message with DLGC_WANTALLKEYS to receive all keyboard input. This technique is not

effective in Windows 3.0 or later.

Method 1

This method involves subclassing the MLE. It is most useful when the edit control is already subclassed for another reason because the amount of additional code is minimal. This method involves three steps:

1. Check for a WM_KEYDOWN message with wParam set to VK_RETURN.

2. Post a WM_CHAR message with wParam set to 0x0a to the edit control.

3. Do not pass the WM_KEYDOWN message to the original window procedure by returning immediately from the subclass procedure.

The following code fragment demonstrates this method:

```
    // In the subclass procedure
    switch (msg)
        {
        case WM_KEYDOWN:
            if (VK_RETURN == wParam)
                {
                PostMessage(hWnd, WM_CHAR, 0x0A, 0L);
                return 0L;
                }
            break;

        ...
        }

    return CallWindowProc(...);
```

The disadvantage to subclassing is that it may be more fragile, with respect to changes in future versions of Windows, than other methods.

Method 2

This method uses the DM_GETDEFID message that Windows sends to a dialog box procedure when the user presses the ENTER key in a dialog box. This method has the additional advantage of easily handling multiple multiline edit controls in the same dialog box and involves three steps:

1. Process WM_COMMAND messages with the HIWORD(lParam) set to EN_SETFOCUS and EN_KILLFOCUS to determine if an MLE has the focus. When EN_SETFOCUS signals that a MLE has the focus, set a static flag to TRUE. When EN_KILLFOCUS signals that a MLE has lost the focus, reset the flag to FALSE.

2. When the dialog procedure receives a DM_GETDEFID message, an edit control has the focus, and the ENTER key is down, post a WM_CHAR message with wParam set to 0x0a to the edit control with the focus.

3. If an edit control had the focus, return TRUE from the dialog function; otherwise, the function must return FALSE. Failing to return FALSE will prevent the ENTER key from behaving properly when an edit control does not have the focus.

The following code fragment demonstrates this procedure:

```
    static fEditFocus;

    switch (msg)
        {
        case WM_COMMAND:
            // ID_EDIT is a multiline edit control
            if (ID_EDIT == wParam)
                {
                if (EN_KILLFOCUS == HIWORD(lParam))
                    fEditFocus = FALSE;

                if (EN_SETFOCUS == HIWORD(lParam))
                    fEditFocus = TRUE;
                }
            else
                ...
            break;

        case DM_GETDEFID:
            /*
             * Check if an edit control has the focus and that
             * the ENTER key is down. DM_GETDEFID may be sent
             * in other situations when the user did not press
             * the ENTER key.
             */
            if (fEditFocus && (0x8000 & GetKeyState(VK_RETURN)))
                {
                PostMessage(hEdit, WM_CHAR, 0x0A, 0L);
                return TRUE;
                }
            break;
        }
    return FALSE;
```

Method 3

Applications that use modeless dialog boxes are required to filter messages through the IsDialogMessage function. The IsDialogMessage function modifies certain messages to implement dialog box behavior. For example, the ENTER key message is modified to generate a WM_COMMAND message with wParam set to IDOK.

An application is, however, free to modify the message before passing it to the IsDialogMessage function. The code example below modifies the WM_KEYDOWN message containing a VK_RETURN to be a EM_REPLACESEL message with a carriage return (CR) and linefeed (LF) combination.

The disadvantage to this method is that it places additional code in the main message loop for the application, slowing the processing of every message. In addition, code in the message loop is far removed from the dialog procedure, and is therefore harder to maintain.

The following code fragment demonstrates this method:

```
    /*
     * hWndEditControl is the handle to the multiline edit control.
     * hWndModeless is the handle to the modeless dialog box.
     */

    while (GetMessage(&msg, NULL, 0, 0))
        {
        if (hWndEditControl == msg.hWnd
                && WM_KEYDOWN == msg.message
                && VK_RETURN == msg.wParam)
            /*
             * Normally, Windows will translate this to IDOK.
             * Perform a custom translation to something more useful
             * (replace selection with a carriage return-linefeed).
             */
            {
            msg.message = EM_REPLACESEL;
            msg.wParam  = 0;
            msg.lParam  = (long)(LPSTR)"\015\012";
            }

        if (!IsDialogMessage(hWndModeless, &msg))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }
```

If the functionality of a modal dialog box is desired in an application, it can be simulated by using a modeless dialog box that disables its parent window when the dialog is created.

Additional reference words: 3.00 3.10 softlib MLEENTER.ZIP
KBCategory:
KBSubcategory: UsrDlgsTabbing

INF: Using a Fixed-Pitch Font in a Dialog Box

Article ID: Q77991

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To use a fixed-pitch font in a dialog box, during the processing of
the dialog box initialization message, send the WM_SETFONT message to
each control that will use the fixed font. The following code
demonstrates this process:

```
   case WM_INITDIALOG:
     SendDlgItemMessage(hDlg, ID_CONTROL, WM_SETFONT,
           GetStockObject(ANSI_FIXED_FONT), FALSE);
     /*
      * Note: This code will specify the fixed font only for the
      * control ID_CONTROL. To specify the fixed font for other
      * controls in the dialog box, additional calls to
      * SendDlgItemMessage() are required.
      */
     break;
```

This information is also outlined in Section 1.4.1 of the TIPS.TXT
file, which is included in the Windows Software Development Kit (SDK).
TIPS.TXT is installed into the Windows development directory (by
default, C:\WINDEV).

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgs

INF: Efficiency of Using SendMessage Versus SendDlgItemMessage

Article ID: Q66944

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The SendDlgItemMessage function is equivalent to obtaining the handle of a dialog control using the GetDlgItem function and then calling the SendMessage function with that handle. The SendDlgItemMessage function therefore takes slightly longer to execute than the SendMessage function for the same message, because an extra call to the GetDlgItem function is required each time the SendDlgItemMessage function is called.

The GetDlgItem function searches through all controls in a given dialog box to find one that matches the given ID value. If there are many controls in a dialog box, the GetDlgItem function can be quite slow.

If an application needs to send more than one message to a dialog control at one time, it is more efficient to call the GetDlgItem function once, using the returned handle in subsequent SendMessage calls. This saves Windows from searching through all the controls each time a message is sent. The SendMessage function should also be used when your application retains handles to controls that receive messages.

However, if your application needs to send one message to many controls, such as sending WM_SETFONT messages to all the controls in a dialog, then the SendDlgItemMessage function will save code in the application because a call to the GetDlgItem function is not made for each control.

Note that if the message sent to a control may result in a lengthy operation (such as sending the LB_DIR message to a list box), then the overhead in the GetDlgItem call is negligible. Either the SendDlgItemMessage or SendMessage can be used, whichever is more convenient.

Additional reference words: 3.00 3.10 3.x send mess dlgitem
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Expanding the Size of a Dialog Box

Article ID: Q35499

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The EXPAND file in the Software/Data Library demonstrates how to create a dialog box that expands to provide additional controls when the user clicks on a button control.

EXPAND can be found in the Software/Data Library by searching on the word EXPAND, the Q number of this article, or S12027. EXPAND was archived using the PKware file-compression utility.

The remainder of this article describes the algorithm demonstrated in the EXPAND file.

More Information:

To use an expanding dialog box in an application, perform the following two preparation steps:

1. Design the entire dialog box and determine what controls will not be initially visible.

2. Disable these hidden controls in the dialog box template by including the WS_DISABLED style bit. This prevents the TAB key or the appropriate mnemonic key from giving the focus to one of the hidden controls.

When the user clicks the "expand" button, perform the following five steps:

1. Disable the expand button using EnableWindow(hWnd, FALSE).

2. Enable the previously hidden controls using EnableWindow(hWnd, TRUE).

3. Use SetFocus() to place the focus on the appropriate control.

4. Use the following function to retrieve the current position and size of the dialog:

   ```
   GetWindowRect(hWndDlg, (LPRECT)&r);
   ```

5. Use the following function to expand the dialog to its new size

   ```
   MoveWindow(hWndDlg, r.left, r.top, r.right - r.left,
                       (r.bottom - r.top) + change, TRUE);
   ```

where "change" is the difference between the small size and the large size.

Note that the MoveWindow() function can be used to expand a window in any direction.

Additional reference words: 2.x 3.00 softlib
KBCategory:
KBSubcategory: UsrDlgs

PRB: Dialog Box with Edit Control Cannot Be Created

Article ID: Q78543

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

The CreateDialog() or DialogBox() function fails to create a dialog box with an edit control when at least 10 dialog boxes, each with one or more edit controls, exist anywhere in the system.

CAUSE

When Windows creates a dialog box with one or more edit controls, it calls the GlobalAlloc() function to allocate a block of global memory. This memory is used as a data segment to hold the text buffers for the edit controls. Due to a limitation of real mode, Windows requires that the handle to the data segment is less than 0x2000. For this reason, Windows maintains a cache of 10 handles less than 0x2000 to use if the handle returned by GlobalAlloc does not meet this requirement. If GlobalAlloc returns a handle greater than 0x2000 and all 10 of the cached handles are in use, then Windows fails the CreateDialog or DialogBox call.

RESOLUTION

Specify the DS_LOCALEDIT style in the dialog box template, or create the edit control explicitly with CreateWindow.

When an edit control has the DS_LOCALEDIT style, Windows does not allocate global memory for the control's data segment. Instead, the edit control uses the data segment of the hInstance variable specified when the dialog box was created. If the application's hInstance is passed to CreateDialog or DialogBox, the edit control uses the application's data segment.

If the edit control is explicitly created with CreateWindow, it will use the data segment pointed to by the hInstance passed to CreateWindow.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Sample Code Demonstrates Using Dialog Box Templates

Article ID: Q78953

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

DIALOG is a file in the Software/Data Library that demonstrates how to create a dialog box template to use with the CreateDialogIndirect() and DialogBoxIndirect() functions.

DIALOG includes a collection of functions that can be included in another application to assist in the process of creating the dialog. DIALOG also includes a test program that can be used to display the dialog on the screen.

DIALOG can be found in the Software/Data Library by searching on the word DIALOG, the Q number of this article, or S12026. DIALOG was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Scrolling Dialog Box Sample Code in Software Library
Article ID: Q75336

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The procedure to define a scrolling dialog box is quite straightforward. It is done by specifying the WS_VSCROLL and WS_HSCROLL window styles. The dialog procedure must set the scroll range and process horizontal and vertical scrolling messages. Windows will process the WM_PAINT message on behalf of the dialog box.

SCROLDLG.ZIP, a file in the Software/Data Library, contains code to a sample application that displays a scrolling dialog box. SCROLDLG can be found in the Software/Data Library by searching on the keyword SCROLDLG, the Q number of this article, or S13138. SCROLDLG was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Sample Code Demonstrates an Application Sign-On Screen
Article ID: Q79976

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

Many applications have a "sign-on" screen that is displayed while the application is loading and initializing. This screen remains visible for some set amount of time or until the user performs some action that interacts with the application. At that point, the sign-on screen disappears and the application may be used as normally.

SIGNON can be found in the Software/Data Library by searching on the keyword SIGNON, the Q number of this article, or S13265. SIGNON was archived using the PKware file-compression utility.

The sign-on screen can be implemented by using a modeless dialog box and performing some additional processing in the application's main message loop. After the main window is created and displayed, a modeless dialog box is created and displayed. This modeless dialog box usually contains the company name and application copyright information. The application then waits for any action indicating that the user is ready to work, such as clicking the mouse or typing on the keyboard. Additionally, the application could start a timer and remove the sign-on screen after a set amount of time (five seconds, for example).

Additional reference words: 3.00 3.10 3.x softlib
KBCategory:
KBSubcategory: UsrDlgsModeless

INF: Overlapping Controls Are Not Supported by Windows

Article ID: Q79981

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary :

Child window controls should not be overlapped in applications for the Windows operating system. When one control overlaps another control, or another child window, the borders shared by the controls may not be drawn properly. Overlapping control may confuse the user of the application because clicking the mouse in the common area may not activate the control that the user intended to activate. This behavior is a consequence of the way that Windows is designed.

More Information :

The following example illustrates the painting problems caused by the ambiguity of overlapping borders. Consider three edit controls, called A, B and C, which overlap each other and an enclosing child window D as shown below:

```
     ____________________________________________
     | | A            | B            | C            | |
     | ---------------------------------------- |
     |                 D                          |
     --------------------------------------------
```

Assume that control B has the focus. If this set of controls is covered by another window, which is subsequently moved away, Windows will send a series of client and nonclient messages to each of the controls and to the enclosing child window. The result of these messages may appear as the illustration below, where the portion of window B's border that overlapped with part of window D's border is missing:

```
     _______________             ________________
     | | A            | B            | C            | |
     | ---------------------------------------- |
     |                 D                          |
     --------------------------------------------
```

Repainting problems related to overlapping controls may vary depending on the version of Windows used.

Additional reference words: 1.x 2.x 3.00 CS_PARENTDC WS_CLIPCHILDREN WS_CLIPSIBLINGS WM_NCPAINT WM_PAINT
KBCategory:
KBSubcategory: UsrDlgsRare/misc

PRB: BS_GROUPBOX-Style Child Window Background Painting Wrong

Article ID: Q79982

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS
When a BS_GROUPBOX style window is created, its background does not erase correctly.

CAUSE
The parent window of the BS_GROUPBOX style window has the WS_CLIPCHILDREN style, which prevents the parent window from erasing the group box's background.

RESOLUTION
Subclass the group box window to process the WM_ERASEBKGND message by erasing its background. Listed below is a code fragment to demonstrate this procedure.

More Information:

The WS_CLIPCHILDREN style causes a window to exclude the areas occupied by child windows when the window paints its client area. However, a BS_GROUPBOX style window is a static control that never erases its background. Erasing the background removes any controls or buttons that appear within the group box.

Therefore, when another child window is dropped over a group box and subsequently dragged away, portions of that child window remain visible in the group box's background. This problem does not occur when the parent window does not have the WS_CLIPCHILDREN style.

The following code fragment should be placed in the group box's subclass procedure. This code erases the background of the group box.

```
   case WM_ERASEBKGND:
       {
       HBRUSH  hBrush, hOldBrush;
       HPEN    hPen, hOldPen;
       RECT    rect;
       HDC     hDC;

       hDC = GetDC(hWnd);

       // Obtain a handle to the parent window's background brush.
       hBrush = GetClassWord(ghWnd, GCW_HBRBACKGROUND);
       hOldBrush = SelectObject(hDC, hBrush);
```

```
   // Create a background-colored pen to draw the rectangle
   // borders, where gWindowColor is some globally defined
   // COLORREF variable used to paint the window's background
   hPen = CreatePen(PS_SOLID, 1, gWindowColor);
   hOldPen = SelectObject(hDC, hPen);

   // Erase the group box's background.
   GetClientRect(hWnd, &rect);
   Rectangle(hDC, rect.left, rect.top, rect.right, rect.bottom);

   // Restore the original objects before releasing the DC.
   SelectObject(hDC, hOldPen);
   SelectObject(hDC, hOldBrush);

   // Delete the created object.
   DeleteObject(hPen);

   ReleaseDC(hWnd, hDC);

   // Instruct Windows to paint the group box text and frame.
   InvalidateRect(hWnd, NULL, FALSE);

   // Insert code here to instruct the contents of the group box
   // to repaint as well.

   return TRUE; // Background has been erased.
   }
```

Additional reference words: 3.00 3.10 3.x SR# G911204-26
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: How Dialog Box Functions Return Values Indicate Failure
Article ID: Q66364

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Windows has eight functions that are used to create dialog boxes: four are for modal dialog boxes, and four are for modeless dialog boxes.

Modal dialog boxes are created using the DialogBox, DialogBoxIndirect, DialogBoxParam, and DialogBoxIndirectParam functions. These functions return -1 to signal failure.

Modeless dialog boxes are created with the CreateDialog, CreateDialogIndirect, CreateDialogParam, and CreateDialogIndirectParam functions. These functions returns NULL to signal failure.

The "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0 incorrectly specifies on page 4-44 the failure return value for the CreateDialogParam function to be -1. The documentation should state that the function returns NULL. This documentation error has been corrected on page 91 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" for version 3.1.

More Information:

The DialogBox* functions that create modal dialog boxes return the value specified as the second parameter to the EndDialog function, which is used to end the processing in a modal dialog box.

The DialogBox* functions cannot return NULL (==FALSE) to indicate an error because returning the TRUE or FALSE values through the EndDialog function might be useful to an application: possibly to indicate that the user selected an OK or Cancel button.

The CreateDialog* functions that create modeless dialog boxes return a window handle when the dialog box is successfully created. The value -1 (0xFFFF) can be a valid window handle and is not sufficient to indicate a situation in which the function failed and a dialog box was not created. Because NULL is the only invalid window handle, it is used as a return value to indicate the failure of a CreateDialog* function. The NULL return value indicates failure for all Windows functions that return a HANDLE data type.

Additional reference words: 3.00 3.10 docerr param indirect
KBCategory:
KBSubcategory: UsrDlgsRare/misc

PRB: Dialog Box and Parent Window Disabled

Article ID: Q11337

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

When an application uses one of the DialogBox family of functions to create a modal dialog box, both the parent window and the dialog box are disabled (unable to accept keyboard or mouse input).

CAUSE

In the application's resource file, the dialog box resource has the WS_CHILD style.

RESOLUTION

To avoid this problem, use the WS_POPUP style instead of the WS_CHILD style.

More Information:

When an application creates a modal dialog box using one of the DialogBox family of functions, Windows disables the dialog box's parent window. If the parent window has any child windows, the child windows are also disabled.

An application can use the WS_CHILD style for dialog boxes created by one of the CreateDialog family of functions. However, problems and inconsistencies arise if the application uses the IsDialogMessage function to process dialog box input for either the parent or the child.

Additional reference words: 2.00 2.03 2.10 3.00 3.10 2.x
CreateDialogIndirect CreateDialogIndirectParam CreateDialogParam
DialogBoxIndirect DialogBoxIndirectParam DialogBoxParam

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsModal

INF: Possible Causes of Dialog Box Creation Failure
Article ID: Q80843

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary :

The dialog box creation routines (CreateDialog(), CreateDialogParam(), CreateDialogIndirect(), CreateDialogIndirectParam(), DialogBox(), DialogBoxParam(), DialogBoxIndirect(), and DialogBoxIndirectParam()) can fail for several reasons. When one of these functions fails, the dialog box is not displayed on the screen. Of the causes for dialog box creation failure, this article discusses nine, which are listed below, and provides a resolution or explanation:

1. Application runs out of file handles.

2. One or more text strings in a dialog resource starts with the character represented by the value 255 (0xFF).

3. Several small compiled resource (RES) files are combined using the MS-DOS command "COPY /b".

4. A dialog box with an edit control cannot be created when at least ten dialog boxes, each with one or more edit controls, are open simultaneously.

5. The dialog resource is not included in the RC file.

6. Insufficient system resources.

7. In a dialog registered with a private dialog class, the dialog procedure does not return the value returned from DefDlgProc as its return value.

8. Wrong HINSTANCE value used.

9. A dialog box with an edit control or a combo box with an edit control must have a HEAPSIZE statement with a value > 0 in the .DEF file.

More Information:

The order in which the causes are listed below does not provide any indication of how often each cause occurs.

Cause 1: Application runs out of file handles.

Resolution 1: Use the SetHandleCount() function to open more file handles. For more information on this procedure, query this knowledge base on the words:

   prod(winsdk) and resources and SetHandleCount
Explanation 1: Windows requires a file handle to load a resource. Because each dialog box creation routine loads dialog resources, an application must have at least one file handle available for that purpose.

Cause 2: One or more text strings in a dialog resource start with the character represented by the value 255 (0xFF).

Resolution 2: For an explanation of two methods to work around this cause, query this knowledge base on the words:

   prod(winsdk) and string and resource and 255
Explanation 2: Windows uses the number 255 to indicate that a resource is represented by an ordinal value instead of by a string name. During the process of parsing the resource that contains one of these characters, Windows incorrectly skips the next two bytes and treats the new position in the resource as the next piece of data.

Cause 3: Several small compiled resource (RES) files are combined using the MS-DOS command "COPY /b".

Resolution 3: Use the #include directive to combine the files at the source level. If the Windows Resource Compiler cannot handle the large RC file, consider storing the resources in a resource-only DLL. For more information, query this knowledge base on the words:

   prod(winsdk) and combined and res
Explanation 3: The Resource Compiler shipped with version 3.0 of the Windows Software Development Kit (SDK) has been enhanced to handle resource files much larger than its Windows 2.x counterpart. The format of the compiled resource files in Windows 3.0 does not support concatenation of RES files.

Cause 4: A dialog box with an edit control cannot be created when at least ten dialog boxes, each with one or more edit controls, are open simultaneously.

Resolution 4: Specify the DS_LOCALEDIT style in the dialog box template, or create the edit control explicitly with the CreateWindow function.

Explanation 4: For more information, query this knowledge base on the words:

   prod(winsdk) and dialog and GlobalAlloc and fails
Cause 5: The dialog resource is not included in the RC file.

Resolution 5: Use RCINCLUDE to include the dialog resource in the RC file.

Explanation 5: Include the DLG file created by the Dialog Editor into the RC file so the Resource Compiler can add the dialog resource to

the executable file.

Cause 6: Insufficient system resources.

Resolution 6: Verify that system resources are not lost because an application does not delete objects that it creates. For more information, query this knowledge base on the words:

```
   prod(winsdk) and lost and heapwalk
```

The article "Careful Windows Resource Allocation and Cleanup Improves Application Hygiene," in the September 1991 issue of the "Microsoft Systems Journal" discusses this issue further.

Explanation 6: Unless each application frees all the resources that it allocates, eventually Windows does not have enough memory to create the controls and/or the dialog box itself.

Cause 7: In a dialog registered with a private dialog class, the dialog procedure does not return the value returned from DefDlgProc as its return value.

Resolution 7: Whenever the private-class dialog procedure passes an unprocessed message to DefDlgProc, the dialog procedure must propagate the value returned by DefDlgProc.

Explanation 7: When a private-class dialog procedure passes an unprocessed message to DefDlgProc, it must return the value returned from DefDlgProc. This behavior differs from that of a default-class dialog procedure, which usually returns FALSE outside the message switch when it does not process a message. If a private-class dialog procedure is implemented in this way, and the procedure does not process the WM_NCCREATE message, it will return FALSE. The CreateWindow function sends the WM_NCCREATE message to create the non-client area of the dialog window. CreateWindow treats a response of FALSE from the dialog procedure as failure and returns a NULL handle to the application.

Cause 8: Wrong HINSTANCE value used.

Resolution 8: Specify the application's instance handle.

Explanation 8: The dialog box creation routine must specify the application's instance because the dialog template is stored in the application instance.

Cause 9: A dialog box with an edit control or a combo box with an edit control must have a HEAPSIZE statement with a value > 0 in the .DEF file.

Resolution 9: Set the value of HEAPSIZE to > 0 in your .DEF file.

Explanation 9: Windows allocates memory for edit controls out of your local heap. If you do not have a HEAPSIZE statement, it cannot create

the edit control.

Additional reference words: fails modal modeless
KBCategory:
KBSubcategory: UsrDlgs

INF: Windows Dialog-Box Style DS_ABSALIGN

Article ID: Q11590

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Windows dialog-box style DS_ABSALIGN (used in WINDOWS.H) means "Dialog Style ABSolute ALIGN." Specifying this style in the dialog template tells Windows that the dtX and dtY values of the DLGTEMPLATE struct are relative to the screen origin, not the owner of the dialog box. When this style bit is not set, the dtX and dtY fields are relative to the origin of the parent window's client area.

Use this term if the dialog box must always start in a specific part of the display, no matter where the parent window is on the screen.

Additional reference words: 2.00 2.03 2.10 3.00 2.x
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Creating a Multiple Line Message Box

Article ID: Q67210

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Message boxes are used to provide information to the user of an application. Error messages and warnings are also provided through message boxes. This article provides details on using message boxes in applications.

More Information:

Message boxes are modal windows. When an application displays an application modal message box, which is the default message box type, the user cannot interact with any part of that application until the message box has been dismissed. However, the user may use the mouse or keyboard to activate another application and interact with it while the message box is displayed. Certain critical errors that may affect all of Windows are displayed in system modal message boxes. Windows will not perform any operations until the error condition is acknowledged and the system modal message box is dismissed.

There are times where it is necessary to display a long message in a message box. Windows does this when you start a DOS application that uses graphics from inside a DOS window. To break a message into many lines, insert a newline character into the message text. Here is a sample MessageBox() call:

```
   MessageBox(hWnd, "This is line 1.\nThis is line 2.", "App",
       MB_OK | MB_ICONQUESTION);
```

If the text of a message is too long for a single line, Windows will break the text into multiple lines.

System modal message boxes treat the newline character as any other. A newline character is displayed as a black block in the text. Because system modal message boxes are designed to work at all times, even under extremely low memory conditions, it does not provide the ability to display more than one line of text.

Additional reference words: 2.x 2.00 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Obtaining Index of Focused Item in Multi-Select List Box

Article ID: Q72042

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Sending an LB_GETCURSEL to a single selection list box returns the index of the currently selected item. However, this message is not designed to be used with multiple selection list boxes. The LB_GETCARETINDEX message is used to determine which item has the focus in a multiple selection list box.

The LB_GETCARETINDEX message exists in Windows version 3.00 but it is not listed in the documentation. It will be documented in the next version of Windows.

To use LB_GETCARETINDEX with Windows 3.00, add the following line to the code:

```
   #define LB_GETCARETINDEX   (WM_USER+32)
```

LB_GETCARETINDEX returns the index of the item with the focus rectangle (dotted caret) in a multiple selection list box. wParam and lParam are not used.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsIds

INF: Sample Code Demonstrates Creating Dialog Box in DLL

Article ID: Q82170

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

DLLDLG is a file in the Software/Data Library that demonstrates creating a dialog box in a dynamic-link library (DLL). The DLL contains a routine that calls the DialogBox function. After the user types some text into an edit control in the dialog box and chooses the OK button, the routine calls the MessageBox function to display the text.

Because a DLL has only one instance, it is not necessary to call the MakeProcInstance function for the dialog box procedure. The DLLDLG sample demonstrates this by specifying the dialog box procedure's name in the call to the DialogBox function.

The DialogBox function uses the value of its hInstance parameter to locate the dialog box's resource template. For this reason, the hInstance value must point to the module that contains the template. For the DLLDLG sample, the hInstance value passed to the DialogBox function is the value of the hInstance parameter passed to LibMain.

DLLDLG can be found in the Software/Data Library by searching on the word DLLDLG, the Q number of this article, or S13370. DLLDLG was archived using the PKware file-compression utility.

Additional reference words: 3.00 3.10 3.x softlib DLLDLG.ZIP
KBCategory:
KBSubcategory: UsrDlgsIndll

INF: Do Not Use MB_NOFOCUS Flag with MessageBox Function
Article ID: Q87341

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In the Microsoft Windows graphical environment, an application can use the MessageBox function to create, display, and operate a message box window. The fourth parameter of the MessageBox function is fuStyle, which has the UINT data type.

The fuStyle parameter specifies the message box style; it can be a combination of the MB_* values, including MB_OK, MB_ICONHAND, MB_SYSTEMMODAL, and so on. These values are defined in the WINDOWS.H header file provided with the Microsoft Windows Software Development Kit (SDK).

WINDOWS.H defines MB_NOFOCUS, which is not listed in the SDK documentation (printed or electronic). In the book "Programming Windows 3" by Charles Petzold (Microsoft Press), page 440 states that when the MB_NOFOCUS flag is specified, Windows creates the message box but does not give it the focus. This statement is incorrect.

MB_NOFOCUS is not one of the standard MB_* values and it cannot be used in the fuStyle parameter. Although MB_NOFOCUS is used internally by Windows, it is not passed to or accepted from an application.

Additional reference words: 3.10 docerr
KBCategory:
KBSubcategory: UsrDlgsMsgbox

INF: Device Independent Way to Use Dialog Box as Main Window
Article ID: Q68556

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A modeless dialog box can act as the main window of a Windows application. Two advantages of this approach are:

1. The main window of the application can be created in the Dialog Editor.

2. Focus control is managed by Windows as it is for a dialog box.

The Software Library contains a file that demonstrates the necessary principles. DLGMAIN can be found in the Software/Data Library by searching on the word DLGMAIN, the Q number of this article, or S12882. DLGMAIN was archived using the PKware file-compression utility.

More Information:

The information below outlines the necessary modifications to the code in GENERIC, a sample application provided with the Windows Software Development Kit (SDK) version 3.0, that will result in a Windows application that uses a modeless dialog box as its main window.

To GENERIC.C, make the following changes:

```
//  1. Create a global variable to hold the window handle of the
//     dialog box.

   HWND hDlgMain;


//  2. In the WinMain() procedure, declare a FARPROC variable. This
//     variable holds the procedure-instance address for the dialog
//     procedure.

   FARPROC lpfn;


//  3. Modify the CreateWindow() call in InitInstance() to specify 0
//     (zero) for nWidth and nHeight when the main window is created.
//     Do this so that the main window is not shown until the size of
//     the dialog box is known.

   hWnd = CreateWindow("GenericWClass", "Dialog as Main Window",
             WS_OVERLAPPED | WS_CAPTION | WS_MINIMIZEBOX | WS_SYSMENU,
             CW_USEDEFAULT, 0, 0, 0, NULL, NULL, hInstance, NULL);
```

```
//  4. In WinMain(), create a procedure-instance for the dialog
//     procedure.

   lpfn = MakeProcInstance(Dialog1, hInst);


//  5. In WinMain, create the dialog box using CreateDialog().

   hDlgMain = CreateDialog(hInst, "DIA1", hWnd, lpfn);


//  6. Add code to WinMain() to show the dialog box.

   ShowWindow(hDlgMain, SW_SHOW);


//  7. Windows sends messages to the modeless dialog box through the
//     program's main message queue. To process messages intended for
//     the dialog box, modify message queue processing in WinMain() as
//     follows:

   while (GetMessage(&msg,   // Message structure
                     NULL,   // Handle of window receiving message
                     NULL,   // Lowest message to handle
                     NULL))  // Highest message to handle
   {
      if (hDlgMain == NULL || !IsDialogMessage(hDlgMain, &msg))
      {
      TranslateMessage(&msg); // Translate virtual key codes
      DispatchMessage(&msg);  // Dispatches message to window
      }
   }
   return msg.wParam;     /* Returns the value from PostQuitMessage */


//  8. When the application receives a WM_SETFOCUS message, it must
//     set the input focus to the control that acts as the
//     application's main window. Adding the following code to
//     MainWndProc() processes the WM_SETFOCUS message as required:

   case WM_SETFOCUS:
      SetFocus(hDlgMain);
      break;
```

In the dialog function, make the remaining changes:

```
//  9. In the dialog function, declare a variable of type RECT to
//     store the bounding rectangle of the modeless dialog box.

   RECT rect;


// 10. Retrieve the dimensions of the dialog box. This information
//     will be used to calculate the size of the parent window.
```

```
   case WM_INITDIALOG:
      GetWindowRect(hDlg, &rect);


// 11. Size the parent window to be the same size as the dialog box.

   SetWindowPos(GetParent(hDlg), // Get the window handle of the
                                 // parent of the dialog box.

     NULL,                       // Identifies a window that in the
                                 // window-manager's list will
                                 // precede the positioned window.
                                 // This is not used for this example.

     0,                          // Specify the x-coordinate of the
                                 // window's upper-left corner.

     0,                          // Specify the x-coordinate of the
                                 // window's upper-left corner.

     rect.right - rect.left,     // Based on the values returned by
                                 // GetWindowRect, calculate the new
                                 // width of the main window.

     rect.bottom - rect.top + GetSystemMetrics(SM_CYCAPTION)
                            + GetSystemMetrics(SM_CYMENU),
                                 // Based on the values returned by
                                 // GetWindowRect, the height of the
                                 // caption bar, and the height of
                                 // the menu bar, calculate the new
                                 // height of the main window.

     SWP_NOMOVE | SWP_NOZORDER);
                                 // Retain current position and
                                 // ordering.


// 12. When the application receives a WM_CLOSE message, close the
//     modeless dialog box and post a WM_CLOSE message to the main
//     window.

   case WM_CLOSE:
      DestroyWindow(hDlg);  // Close the modeless dialog box using
                            // DestroyWindow.
      hDlgMain = NULL;      // Set the handle of the modeless dialog
                            // box to null.
      PostMessage(GetParent(hDlg), WM_CLOSE, NULL, NULL);
      break;
```

Make one change to GENERIC.RC: Modify the dialog box template to specify the additional style WS_CHILD. Doing so will ensure that the dialog box moves with the parent window as the parent is moved.

Additional reference words: 3.00 softlib DLGMAIN.ZIP
KBCategory:

KBSubcategory: UsrDlgsMainwind

INF: Default/Private Dialog Classes, Procedures, DefDlgProc

Article ID: Q68566

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The information below explains the differences between default and private dialog classes, their associated dialog procedures, and using the DefDlgProc function.

This information is organized as a comparison between private and default dialog classes, covering class registration, dialog templates, dialog creation, and message processing.

Note that the source to the DefDlgProc function is provided with the Windows Software Development Kit (SDK) version 3.0. The code is supplied on the Sample Source 2 disk in the DEFDLG.C file. By default, DEFDLG.C is placed into the \WINDEV\DEFPROCS directory if the SDK installation program copies the sample source code as part of the SDK installation process.

There are many functions and macros used in DefDlgProc that are internal to Windows and cannot be used by applications. No additional information is available on these functions and macros.

More Information:

All dialog classes are window classes, just as all dialog boxes are windows. All dialog classes must declare at least DLGWINDOWEXTRA in the cbWndExtra field of the WNDCLASS structure before the dialog class is registered. The Windows Dialog Manager uses this area to store special information for dialog boxes.

The default dialog class is registered by Windows at startup. The window procedure for this class is known as DefDlgProc, which is located in Windows's USER module. DefDlgProc calls the application-provided dialog function, which returns TRUE if it processes a message completely or FALSE if DefDlgProc should process the message further.

If an application registers a private dialog class, it provides a window procedure for the dialog box. The window procedure is the same as that for any other application window and returns a LONG value. Messages that are not processed by this window function are passed to DefDlgProc.

Dialog Class Registration

Windows registers the default dialog class, which is represented by the value 0x8002. Windows uses this class when an application creates

a dialog box using the DialogBox or CreateDialog functions, but specifies no class in the dialog resource template.

To use a private dialog class, the application must specify the fields of a WNDCLASS structure and call RegisterClass. This is the same procedure that Windows uses to register the default dialog class.

In either case, the value in the cbWndExtra field of the WNDCLASS structure must contain a value of at least DLGWINDOWEXTRA. These bytes are used as storage space for dialog-box specific information, such as which control has the focus and which button is the default.

When a dialog class is registered, the lpfnWndProc field of the WNDCLASS structure must contain a function pointer. For the default dialog class, this field points to DefDlgProc. For a private class, the field points to application-supplied procedure that returns a LONG (as does a normal window procedure) and passes all unprocessed messages to DefDlgProc.

Dialog Templates

Resource scripts are almost identical whether used with a default or a private dialog class. Dialog boxes using a private class must use the CLASS statement in the dialog template. The name given in the CLASS statement must match the name of class that exists (is registered) when the dialog box is created.

Dialog Creation and the lpfnDlgFunc Parameter

Applications create dialog boxes using the function DialogBox, CreateDialog, or one of the variant functions such as DialogBoxIndirect. The complete list of functions is found on page 1-43 of the "Microsoft Windows Software Development Kit Reference Volume 1."

All dialog box creation calls take a parameter called lpfnDlgFunc, which can either be NULL or the procedure instance address of the dialog box function returned from MakeProcInstance. When the application specifies a private dialog class and sets lpfnDlgFunc to a procedure instance address, the application processes each message for the dialog box twice. The message processing proceeds as follows:

1. Windows calls the dialog class procedure to process the message. To process a message in the default manner, this procedure calls DefDlgProc.

2. DefDlgProc calls the procedure specified in the dialog box creation call.

The procedure specified in lpfnDlgFunc must be designed very carefully. When it processes a message, it returns TRUE or FALSE and does not call DefDlgProc. These requirements are the same as for any other dialog procedure.

Using a dialog procedure in conjunction with a private dialog class can be very useful. Processing for the private dialog class can be generic and apply to a number of dialog boxes. Code in the dialog procedure is specific to the particular instance of the private dialog class.

Dialog Message Processing

In dialog boxes with the default class, the application provides a callback dialog function that returns TRUE or FALSE, depending on whether or not the message was processed. As mentioned above, DefDlgProc, which is the window procedure for the default dialog class, calls the application's dialog function and uses the return value to determine whether it should continue processing the message.

In dialog boxes of a private class, Windows sends all messages to the application-provided window procedure. The procedure either processes the message like any other window procedure or passes it to DefDlgProc. DefDlgProc processes dialog-specific functions and passes any other messages to DefWindowProc for processing.

Some messages are sent only to the application-supplied procedure specified in the call to CreateDialog or DialogBox. Two examples of functions that Windows does not send to the private dialog class function are WM_INITDIALOG and WM_SETFONT.

Additional reference words: 3.00 MICS3 R1.7
KBCategory:
KBSubcategory: UsrDlgsPrivdlgs

INF: Using the WM_GETDLGCODE Message

Article ID: Q83302

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Windows sends a WM_GETDLGCODE message to controls in a dialog box or in a window where the IsDialogMessage function handles keyboard input. Generally, an application processes the WM_GETDLGCODE message to prevent Windows from performing default processing in response to keyboard messages. The WM_KEYDOWN, WM_SYSCHAR, and WM_CHAR messages are examples of keyboard messages.

This article discusses the various codes that make up the value returned from the WM_GETDLGCODE message.

More Information:

Windows sends a WM_GETDLGCODE message to a control for the following three reasons:

- To determine whether the control will process a particular type of input.

- To determine whether the text contents of an edit control are selected when, as a result of the user pressing the TAB key, the edit control receives the input focus.

- To determine the type of a button.

The following text documents the values of the wParam and lParam parameters for the WM_GETDLGCODE message:

Parameter	Description
wParam	Not used.
lParam	If lParam is not NULL, it is a far pointer to an MSG structure that contains a message that is being sent to the control. Windows versions 3.0 and 3.1 use lParam only to send keyboard input to the control. Keyboard messages include WM_KEYDOWN, WM_SYSCHAR, and WM_CHAR. Future versions of Windows may use lParam to send other message types to controls.

The window procedure for each predefined control returns an appropriate value in response to a WM_GETDLGCODE message. The value is one or more of the following codes, combined with the Boolean OR operator:

Code	Meaning
DLGC_BUTTON	Control is a button (of any kind).
DLGC_DEFPUSHBUTTON	Control is a default push button.
DLGC_HASSETSEL	Windows will send an EM_SETSEL message to the control to select its contents.
DLGC_RADIOBUTTON	Control is an option (radio) button.
DLGC_STATIC	Control is a static control.
DLGC_UNDEFPUSHBUTTON	Control is a push button but not the default push button.
DLGC_WANTALLKEYS	Control processes all keyboard input.
DLGC_WANTARROWS	Control processes arrow keys.
DLGC_WANTCHARS	Control processes WM_CHAR messages.
DLGC_WANTMESSAGE	Control processes the message in the MSG structure that lParam points to.
DLGC_WANTTAB	Control processes the TAB key.

The return codes above can be used by user-defined controls or, in a subclass procedure, to modify the behavior of predefined controls. To subclass a control, call the predefined control's window procedure first, then modify the necessary bits in the return code.

DLGC_WANTALLKEYS, DLGC_WANTARROWS, DLGC_WANTCHARS, DLGC_WANTMESSAGE, and DLGC_WANTTAB Return Codes

When a control processes the WM_GETDLGCODE message and the value it returns has one of the DLGC_WANT* bits set, the control will process the specified message type and Windows will not do any default processing for messages of the specified type.

For example, the code returned by a list box includes DLGC_WANTARROWS to indicate that the list box processes arrow keys. When a list box has the focus and the user presses a DOWN ARROW key, Windows sends a WM_GETDLGCODE message to the list box. Because the return value includes the DLGC_WANTARROWS code, Windows allows the list box to process the arrow keystroke and performs no further processing. If the return value did not include the DLGC_WANTARROWS code, Windows would continue processing the arrow keystroke and would change the focus to the next control in the current control group.

As another example, the value returned by an edit control includes the DLGC_WANTCHARS code while the value returned by a button does not. Consequently, if a button has the focus, and the user types a valid mnemonic character, Windows sets the focus to the control in the

dialog box that corresponds to the mnemonic. (If a control has a mnemonic character, it is underlined in the control's label.) If an edit control has the focus and the user types a mnemonic character, however, Windows does not change the input focus because the edit control processes the resulting WM_CHAR message and Windows does not perform its default processing for a mnemonic character.

DLGC_WANTMESSAGE Code

A control returns a value that includes the DLGC_WANTMESSAGE code after it processes the message sent through the lParam that accompanies the WM_GETDLGCODE message. The DLGC_WANTMESSAGE code indicates that the application does not want default processing for the message to continue. The messages sent to the control include WM_KEYDOWN, WM_SYSCHAR, and WM_CHAR. Future versions of Windows could send other messages to controls using this mechanism.

The following code provides an example of processing the WM_GETDLGCODE message in a control's subclass procedure. In the example, the user presses the "X" key to select a check box and presses the "O" key to clear the check box:

```
   case WM_GETDLGCODE:
      // Call the check box window procedure first
      lRet = CallWindowProc(lpCheckProc, hWnd, wMessage, wParam,
                            lParam);

      // If lParam points to an MSG structure
      if (lParam)
      {
         lpmsg = (LPMSG)lParam;
         if (lpmsg->message == WM_CHAR)
         {
            if (lpmsg->wParam == 'x' || lpmsg->wParam == 'X')
            {
               // Select the check box when user presses "X"
               SendMessage(hWnd, BM_SETCHECK, TRUE, 0);
               lRet |= DLGC_WANTMESSAGE;
            }
            else if (lpmsg->wParam == 'o' || lpmsg->wParam == 'O')
            {
               // Clear the check box when user presses "O"
               SendMessage(hWnd, BM_SETCHECK, FALSE, 0);
               lRet |= DLGC_WANTMESSAGE;
            }
         }
      }
      return lRet;
```

When a check box control's subclass procedure includes the code above, Windows performs no further processing for WM_CHAR messages for the X, x, O, and o characters because the value returned from WM_GETDLGCODE includes the DLGC_WANTMESSAGE code. In the example above, the control could have returned DLGC_WANTCHARS instead of DLGC_WANTMESSAGE because the WM_CHAR message is the only message processed by the control.

DLGC_HASSETSEL Code

An edit control returns a value that includes the DLGC_HASSETSEL code to indicate that Windows should select all the text in an edit control when the control receives the input focus through the tabbing sequence.

For example, when a control in a dialog box receives the focus because the user pressed the TAB key, Windows sends a WM_GETDLGCODE message to the control. If the value returned from the edit control includes the DLGC_HASSETSEL code, the edit control indicates that all text in the edit control should be selected. Consequently, Windows sends an EM_SETSEL message to the control to select all its contents.

An application can alter this behavior and prevent the contents from being selected when the control receives the focus through tabbing, by subclassing the edit control and removing the DLGC_HASSETSEL code from its return value. Note that the subclassing code below does not change any other bits in the return value.

```
   // In the subclass procedure
   case WM_GETDLGCODE:
      // Call the original edit control window procedure
      lRet = CallWindowProc(lpEditProc, hWnd, wMessage, wParam,
                            lParam);

      // Clear the DLGC_HASSETSEL bit from the return value
      lRet &= ~DLGC_HASSETSEL;

      return lRet;
```

DLGC_BUTTON, DLGC_DEFPUSHBUTTON,
DLGC_UNDEFPUSHBUTTON, DLGC_RADIOBUTTON, DLGC_STATIC Codes

These codes are used to determine a control's attributes.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrDlgsKeysinput

INF: Sample Code Demonstrates Using Private Dialog-Box Class

Article ID: Q83365

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

PVTDLG is a file in the Software/Data Library that demonstrates using a private dialog-box class in an application. The PVTDLG application demonstrates using a private dialog-box class by itself and in conjunction with another dialog box procedure. PVTDLG registers a private dialog-box class that specifies a number of additional window extra bytes. The sample stores the colors to paint each window in the allocated space.

PVTDLG can be found in the Software/Data Library by searching on the word PVTDLG, the Q number of this article, or S13377. PVTDLG was archived using the PKware file-compression utility.

More Information:

A private dialog-box class is a technique that allows one callback procedure to process messages that are common to several different dialog boxes. The private dialog-box class encapsulates code that would otherwise be repeated in each dialog procedure.

When Windows has a message for a dialog box that is a member of a private dialog-box class, it calls the private dialog-box class procedure. After the private dialog-box class procedure completes its processing, it must call the DefDlgProc function.

The next step depends on the contents of the lpDialogFunc parameter in the application's call to the CreateDialog or DialogBox functions. If lpDialogFunc is NULL, the DefDlgProc function performs its default processing for the message and processing of that message is complete. If lpDialogFunc is not NULL, DefDlgProc calls the specified procedure. If the dialog box procedure returns FALSE, DefDlgProc performs its default processing for the message. If the dialog box procedure returns any other value, DefDlgProc skips the default processing and returns.

Additional reference words: 3.00 3.10 softlib PVTDLG.ZIP
KBCategory:
KBSubcategory: UsrDlgsPrivDlgs

INF: Types of System Modal Message Boxes
Article ID: Q104959

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

There are two kinds of system modal message boxes:

 - Hard System Modal Message Box

   A hard system modal message box is the only window in the system
   that will receive messages.

 - Soft System Modal Message Box

   All windows owned by a soft system modal message box's task will
   receive messages. No other window in the system will receive
   messages.

MORE INFORMATION
================

If a task wants to bring up a system modal message box and requires that messages be received by its other windows, a soft system modal message box must be used. For example, a task may require that WM_TIMER messages be received by a window it owns even while it displays a system modal message box. On the other hand, a hard system modal message box will prevent any other window in the system from receiving messages. A hard system modal message box is also appropriate in low memory situations because it uses very few system resources.

Both hard and soft system modal message boxes are created using MessageBox(). The MB_ICON* flags indicate the kind of system modal message box to be created.

A hard system modal message box can be created using MessageBox() and by including the following flags in the fuStyle parameter:

```
     MB_SYSTEMMODAL
or   MB_SYSTEMMODAL | MB_ICONHAND
or   MB_SYSTEMMODAL | MB_ICONSTOP
```

(Other flags such as MB_OK, MB_OKCANCEL, and so forth can also be used. However do not use any other MB_ICON* flags).

A soft system modal message box can be created by including the following flags in the fuStyle parameter

```
   MB_SYSTEMODAL | MB_ICON*
```

where MB_ICON* is any MB_ICON* style except MB_ICONHAND or MB_ICONSTOP.

(Other flags such as MB_OK, MB_OKCANCEL, and so forth can also be used. However do not use MB_ICONHAND or MB_ICONSTOP.)

Note that other tasks in the system do not receive any messages when a task puts up a hard or soft system modal message box.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsMsgbox

INF: Clearing a Message Box

Article ID: Q74444

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

During the processing of the MessageBox() function, Windows creates a bitmap to save the part of the screen covered by the message box. Normally, before the MessageBox() function returns, Windows repaints the portion of the screen covered by the message box using the bitmap. In this scenario, when the user clicks on a button to dismiss the message box, the message box disappears immediately.

It is important to note that under low memory conditions, Windows will discard the bitmap. If the bitmap is discarded and a significant amount of processing takes place between the MessageBox() call and painting the application's window, the vestigial image of the message box will remain on the screen during the processing. If the user clicks on this image with the mouse, the underlying window will receive the mouse messages. This can cause unexpected (and possibly undesirable) effects.

To address this problem, call UpdateWindow() immediately after MessageBox(). The parameter to UpdateWindow() should be the parent window of the message box (or of the application's main window if the message box has no parent). This will cause the application to paint the affected window if the bitmap has been discarded. The message box will disappear immediately under all circumstances.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsMsgBox

Using IsDialogMessage to Simulate a Dialog Box
Article ID: Q69077

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There is a sample program in the Software/Data Library named ISDIALOG that demonstrates how to create a window, add controls, and use the IsDialogMessage function to make the window act like a dialog box. This technique can be used in cases where superclassing a dialog box is too costly or not possible.

By using a conventional window and not using the Dialog Manager, the application is free to add or remove dialog functionality as necessary.

ISDIALOG can be found in the Software/Data Library by searching on the keyword ISDIALOG, the Q number of this article, or S12914. ISDIALOG was archived using the PKware file-compression utility.

Additional reference words: MICS3 R1.7 ISDIALOG.ZIP
KBCategory:
KBSubcategory: UsrDlgsIsdialogm

PRB: Disabling All Controls in a Dialog Box Hangs Windows
Article ID: Q84001

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS
   Under Windows 3.0, disabling all controls in a dialog box causes
   Windows to hang.

CAUSE
   The Dialog Manager enters an infinite loop searching for an enabled
   control that can receive the input focus.

RESOLUTION
   Modify the design of the dialog box to include a hidden control
   that is always enabled.

This situation does not occur under Windows 3.1.

More Information:

An application might encounter conditions under which all controls in
a dialog box should be disabled. For example, an application might
disable an edit control to prevent the user from entering text under
certain circumstances. If conditions occur such that the application
disables all controls in a dialog box, Windows will hang if the user
uses the keyboard to move the input focus between controls.

To address this situation, modify the dialog box template to include
an enabled owner-draw button at coordinates (0, 0). When any controls
in the dialog box are enabled, disable the button to prevent it from
receiving the input focus. When the application disables the last
"regular" control, enable the button. Through this procedure, at least
one control is always enabled.

If the application ignores the WM_DRAWITEM messages that draw the
owner-draw button, the button control is invisible to the user.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsIds

INF: Sample Code Demonstrates How to Add Menus to a Dialog Box

Article ID: Q84129

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

DBMENU is a file in the Software/Data Library that demonstrates how to add menus to a dialog box. Placing a menu on a dialog box is not part of the Common User Access (CUA) standard, however, Windows does support it.

This article discusses some requirements that must be observed to use a menu on a dialog box.

DBMENU can be found in the Software/Data Library by searching on the word DBMENU, the Q number of this article, or S13408. DBMENU was archived using the PKware file-compression utility.

More Information:

To properly use a menu on a dialog box, an application must use certain dialog box styles and avoid others. For the user to traverse a dialog box menu using the LEFT and RIGHT ARROW keys, the dialog box must have both the WS_CAPTION and WS_SYSMENU styles. The WS_SYSMENU style adds a system menu to the dialog box. To enable the user to close the dialog box by choosing Close from the system menu, process the WM_COMMAND message in the dialog box procedure. If wParam is set to IDCANCEL, the user has closed the dialog box in this manner.

The dialog box must not use the DS_MODALFRAME style because the menu is not painted properly when this style is specified. If desired, use the WS_BORDER style to place a thin black border around the dialog box. Do not use the WS_CHILD style in any dialog box. Include the WS_VISIBLE style to ensure that the dialog box is drawn on the screen.

Accelerator keys can be used only on modeless dialog boxes created by one of the CreateDialog* functions. An application processes accelerator keys by calling the TranslateAccelerator function in its main message loop. Modal dialog boxes have a "private" message loop and the application cannot insert the necessary call to TranslateAccelerator.

An application can simulate a modal dialog box with a modeless dialog box by calling EnableWindow to disable the application's main window once the dialog box is displayed. The application must call EnableWindow to enable the main window before destroying the modeless dialog box.

Do not use the following key combinations as accelerators: CTRL+H,

CTRL+I, and CTRL+M. For more information, query on the following words in the Microsoft Knowledge Base:

prod(winsdk) and ctrl and accelerator and conflict

The following code demonstrates the additional function calls needed in the main message loop for an application to process accelerator keys on a modeless dialog box:

```
// Accelerators for the main window and dialog are in the same
// accelerator table
hAccel = LoadAccelerators(ghInstance, "ACCELTABLE");

while (GetMessage(&msg, NULL, NULL, NULL))
{
   // Determine the destination for this message
   hWndAccel = GetActiveWindow();

   if (!(hWndAccel
         && TranslateAccelerator(hWndAccel, hAccel, (LPMSG)&msg)))
   {
      // Note: If the dialog does not exist, ghModelessDlg is 0
      if (!(ghModelessDlg && IsDialogMessage(ghModelessDlg, &msg)))
      {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
   }
}
```

Additional reference words: 3.00 3.10 softlib DBMENU.ZIP
KBCategory:
KBSubcategory: UsrDlgsRare/misc

PRB: Undesirable Interactions Between Dialog Box Types
Article ID: Q84133

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary :

Some combinations of modal and modeless dialog boxes create undesirable side effects due to their design and implementation. This article describes the side effects and methods to avoid them. The following list of side effects may not be complete, and will be updated as additional side effects are discovered.

SIDE EFFECTS
----

1. Unable to change focus with the keyboard if a modeless dialog box is created as the child of an active modal dialog box.

2. Unable to change focus with the keyboard if a modeless dialog box is owned by an active modal dialog box.

3. Input focus moves to another application if a modeless dialog box is destroyed during processing of the WM_INITDIALOG message of a modal dialog box owned by the application's top-level window.

DLGS is a file in the Software/Data Library that demonstrates the three side effects listed above, explains the causes of each, and demonstrates the method listed below to avoid each problem.

DLGS can be found in the Software/Data Library by searching on the word DLGS, the Q number of this article, or S13410. DLGS was archived using the PKware file-compression utility.

More Information:

Sections A and B below list some characteristics of the design and implementation of modal and modeless dialog boxes. Section C below explains the causes of the side effects listed above and techniques to avoid these side effects.

SECTION A: CHARACTERISTICS OF A MODAL DIALOG BOX

 - A modal dialog box has its own message loop to process messages from the application's queue without involving the application's message loop. This private message loop is active as long as the modal dialog is active.

 - A modal dialog box disables its owner to prevent the owner from processing input. By default, a modal dialog box disables only one

window; the other windows remain enabled and can process user input unless they are explicitly disabled.

SECTION B: CHARACTERISTICS OF A MODELESS DIALOG BOX

- A modeless dialog box does not disable its owner window. Therefore, the user can continue to work with the owner window while the modeless dialog box is displayed.

- A modeless dialog receives its messages through the application's main message loop.

- An application typically calls the IsDialogMessage function to process keyboard input for the modeless dialog box. IsDialogMessage handles changing the focus between controls using the keyboard.

SECTION C: EXPLAINING THE SIDE EFFECTS

Side Effect 1

SYMPTOMS
When a modeless dialog box is created as a child of a modal dialog box, the keyboard cannot be used to change the focus.

CAUSE
The modal dialog box's message loop does not provide the functionality of the IsDialogMessage function.

RESOLUTION
Substitute a modeless dialog box for the modal dialog box. To make the parent modeless dialog box appear modal, disable its owner window in the code to process the WM_INITDIALOG message.

When a modal dialog box is in its message loop, all windows in the application, including each modeless dialog box, receive its messages from the modal dialog box's message loop. However, this message loop does not provide the functionality of IsDialogMessage.

Side Effect 2

SYMPTOMS
When a modeless dialog box is owned by a modal dialog box, the keyboard cannot be used to change the focus.

CAUSE
The modal dialog box's message loop does not provide the functionality of the IsDialogMessage function.

RESOLUTION
Substitute a modeless dialog box for the modal dialog box. To make the parent modeless dialog box appear modal, disable its owner window in the code to process the WM_INITDIALOG message.

Side Effect 3

SYMPTOMS

If a modeless dialog box is destroyed during the processing of the WM_INITDIALOG message for a modal dialog box owned by the application's top-level window, the focus moves to another application.

CAUSE

No window is available to receive the focus.

RESOLUTION

Substitute a modeless dialog box for the modal dialog box, as above. Disable the owner window (to simulate modality) only after the modeless dialog box is destroyed. Then Windows can put the input focus back to the top window until the simulated modal dialog box is displayed.

Windows sends a WM_INITDIALOG message to a modal dialog box just before the dialog is made visible. As part of destroying a window that has the input focus, Windows removes the focus from the window and gives the focus to another window. If the new modal dialog box destroys the modeless dialog box as it processes a WM_INITDIALOG message, the modal dialog box is not yet visible and cannot receive the input focus. Unless the application has other visible windows, the only window that can receive the input focus is the top-level window. However, because the top-level window owns the new modal dialog box, it is disabled and cannot receive the input focus.

Because none of the active application's windows are eligible to receive the input focus, Windows activates another application and gives the input focus to one of its windows. When this other application receives the focus, it moves to the front, over the application that created the modal dialog box. As soon as the original application processes the WM_INITDIALOG message, the modal dialog box is displayed and brought to the front.

The scenario above causes another application to be "sandwiched" between the application's main window and the modal dialog box, which may confuse the user. The following diagram illustrates the visual effect:

```
+-------------------------------------------------------------+
|         Application 1's Main Window                         |
|  +-------------------------------------------------------+  |
|  |       Application 2's Main Window                     |  |
|  |       (on top of application 1's main window)         |  |
|  | +---------------------------------------------------+ |  |
|  | |      Application 1's Modal Dialog                 | |  |
|  | |      (on top of application 2)                    | |  |
|  | |                                                   | |  |
|  | +---------------------------------------------------+ |  |
|  +-------------------------------------------------------+  |
```

+--+

Additional reference words: 3.00 softlib DLGS.ZIP
KBCategory:
KBSubcategory: UsrDlgsRare/misc

PRB: Vertical Bars Displayed in Message Box, Control Text
Article ID: Q84900

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS
When an application runs under Windows 3.0, text in a message box or in a child-window control appears on two lines, as desired. However, under Windows 3.1, the same text appears as one line with two vertical bars at the position where the line break should appear.

CAUSE
The character sequence "\n\r" was used to separate the two lines of text.

RESOLUTION
Modify the text to use the character sequence "\r\n" to break lines of text.

More Information:

In an application developed for the Windows environment, various text strings can contain an embedded carriage return-linefeed pair to display the text on two separate lines. For example, an application can send the following message to a multiline edit control in a dialog box to display the string "This is a test" on two lines with a break between the words "a" and "test":

```
   SetDlgItemText(hDlg, IDC_MEDIT, (LPSTR)"This is a\r\ntest."));
```

Under Windows 3.0, both the "\r\n" and "\n\r" strings create a new line in the output text. However, under Windows 3.1, only the "\r\n" string creates a new line in the output. The "\n\r" string creates the two vertical bars discussed above.

Additional reference words: 3.00 3.10 static new line
KBCategory:
KBSubcategory: UsrDlgsMsgbox

INF: Sample Code Demonstrates Changing Dialog Box Size
Article ID: Q84980

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

The Color dialog box provided by the COMMDLG dynamic-link library (DLL) expands to show additional controls when the user chooses the Define Custom Colors button. The XPANDDLG sample application in the Software/Data Library demonstrates how an application can implement similar behavior.

XPANDDLG can be found in the Software/Data Library by searching on the word XPANDDLG, the Q number of this article, or S13438. XPANDDLG was archived using the PKware file-compression utility.

The XPANDDLG sample implements changing the size of the dialog box through the following three steps:

1. Create the dialog box at its large size.

2. Call the MoveWindow function to size the window to the smaller size when the dialog box procedure receives a WM_INITDIALOG message.

3. To show the remainder of the dialog box on the screen, call MoveWindow again.

Additional reference words: 3.00 3.10 softlib XPANDDLG.ZIP
KBCategory:
KBSubcategory: UsrDlgsMsgbox

INF: Using One IsDialogMessage Call for Many Modeless Dialogs

Article ID: Q71450

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the Windows environment, an application can implement more than one modeless dialog box with a single call to the IsDialogMessage function. This can be done by using the following three-step method:

1. Maintain the window handle to the currently active modeless dialog box in a global variable.

2. Pass the global variable as the hDlg parameter to the IsDialogMessage function, which is normally called from the application's main message loop.

3. Update the global variable whenever a modeless dialog box's window procedure receives a WM_ACTIVATE message, as follows:

   - If the dialog is losing activation (wParam is 0), set the global variable to NULL.

   - If the dialog is becoming active (wParam is 1 or 2), set the global variable to the dialog's window handle.

More Information:

The information below demonstrates how to implement this technique.

1. Declare a global variable for the modeless dialog box's window handle.

```
HWND hDlgCurrent = NULL;
```

2. In the application's main message loop, add a call to the IsDialogMessage function.

```
while (GetMessage(&msg, NULL, 0, 0))
   {
   if (NULL == hDlgCurrent || !IsDialogMessage(hDlgCurrent, &msg))
      {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }
   }
```

3. In the modeless dialog box's window procedure, process the WM_ACTIVATE message.

```
   switch (message)
      {
      case WM_ACTIVATE:
         if (0 == wParam)              // becoming inactive
            hDlgCurrent = NULL;
         else                          // becoming active
            hDlgCurrent = hDlg;

         return FALSE;
      }
```

For more information on the WM_ACTIVATE message, see page 6-47 in the "Microsoft Windows Software Development Kit (SDK) Reference Volume 1."

For details on the IsDialogMessage function, see page 4-266 in the SDK reference, Volume 1.

For details on using a modeless dialog box in an application for the Windows environment, see Chapter 10 of "Programming Windows," second edition, (Microsoft Press) written by Charles Petzold.

Additional reference words: 3.00 MICS3 R1.7 focus
KBCategory:
KBSubcategory: UsrDlgsModeless

INF: Menus Supported in Dialog Boxes w/o DS_MODALFRAME Style

Article ID: Q71499

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows,
   versions 3.0 and 3.1

Dialog boxes in Windows version 3.0 and later suppport menus as long as the DS_MODALFRAME style is absent and the WS_SYSMENU style is present. This behavior will continue to be supported in future versions of Windows.

Dialogs with a modal frame and menu do not work and are not designed to do so.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsModal

INF: GetNextDlg[Group/Tab]Item() Documentation Incorrect

Article ID: Q71501

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the "Windows Software Development Kit Reference Volume 1," the documentation for the GetNextDlgGroupItem() function on page 4-192 and for the GetNextDlgTabItem() function on page 4-193 is incorrect. In both cases, the bPrevious parameter is described incorrectly in the syntax section. The Comments section for both functions is correct.

The bPrevious parameter sets the search direction for GetNextDlgTabItem() and GetNextDlgGroupItem(). Both functions can find the next or the previous control. A nonzero value (TRUE) causes the two functions to search backward while zero (FALSE) causes the functions to search forward.

GetNextDlgGroupItem() searches for controls with the WS_GROUP style.
GetNextDlgTabItem() searches for controls with the WS_TABSTOP style.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Using a Modeless Dialog Box with No Dialog Function

Article ID: Q72136

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When creating a modeless dialog box with the default dialog class, an application normally passes a procedure-instance address of the dialog function to the CreateDialog() function. This dialog function processes messages such as WM_INITDIALOG and WM_COMMAND, returning TRUE or FALSE.

It is acceptable to create a modeless dialog box that uses NULL for the lpDialogFunc parameter of CreateDialog(). This type of dialog is useful when the no controls or other input facilities are required. In this case, using NULL simplifies the programming.

However, the dialog must be closed through some means other than a push button. This might be a timer event. NOTE: A modal dialog box that does not provide a means of closing itself will hang its parent application because control will never return from the DialogBox() function call.

If lpDialogFunc is NULL, no WM_INITDIALOG message will be sent, and DefDlgProc() does not attempt to call a dialog function. Instead, DefDlgProc() handles all messages for the dialog. The application that created the modeless dialog must explicitly call DestroyWindow() to free its system resources.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Dialog Box Placement

Article ID: Q35643

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Adding the following code to the dialog-box routine will move the side of the dialog box to be flush with the left side of the screen. This code will move the dialog box before it is displayed.

```
   RECT r;

   case WM_INITDIALOG:
       GetWindowRect(hDlg,&r);
       MoveWindow (hDlg, 0,r.top,r.right-r.left,r.bottom-r.top , TRUE);
       break;
```

GetWindowRect() will get the current size of the window and the MoveWindow() call will move the window to the left of the screen. As a result, the dialog box will be at the same Y position at which it was originally located.

Additional reference words: 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Context-Sensitive Help in a Dialog Box Through F1

Article ID: Q72219

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In an application developed with Windows version 3.0, to implement the ability to request context-sensitive help by pressing the F1 key in a dialog box, it is necessary to install a message filter.

This article describes the details of installing the message filter. In particular, it describes code that could be added to the Generic sample application (included with the Windows Software Development Kit version 3.0) to install a message filter that would detect F1 keystrokes in the About Generic modal dialog box.

More Information:

Detecting F1 Keystrokes in a Dialog Box

Normally, the child control windows in a dialog box do not pass keystroke messages on to the dialog box window. To detect F1 keystrokes in a dialog box, it is necessary to install a message filter. Page 18-20 of the "Microsoft Windows Software Development Kit Tools" manual describes the method for detecting F1 keystrokes in a dialog box.

Installing a Filter Function

Page 1-65 of the "Microsoft Windows Software Development Kit Reference, Volume 1" describes the method for installing a filter function.

Sample Code

Perform the following six steps to add code to the Generic sample application to provide context-sensitive help in the About Generic modal dialog box:

1. Define a private message that the filter function can post to the application. For example:

```
#define PM_CALLHELP  WM_USER+1000
                                // wParam and lParam can be used to
                                // pass context information
```

2. Define global variables:

```
   FARPROC lpfnFilterProc;        // Our filter function
   FARPROC lpfnOldHook;           // Previous filter function
                                  // in the chain
```

3. Add a filter function to the C-language source file. For example:

```
/*-------------------------    FilterFunc -------------------------*/
//
//   PARAMETERS:
//
//      nCode : Specifies the type of message being processed. It will
//              be either MSGF_DIALOGBOX, MSGF_MENU, or less than 0.
//
//      wParam: specifies a NULL value
//
//      lParam: a FAR pointer to a MSG structure
//
//
//   GLOBAL VARIABLES USED:
//
//      lpfnOldHook
//
//
//   NOTES:
//
//     If (nCode < 0), return DefHookProc IMMEDIATELY.
//
//     If (MSGF_DIALOGBOX==nCode), set the local variable ptrMsg to
//     point to the message structure. If this message is an F1
//     keystroke, ptrMsg->hwnd will contain the HWND for the dialog
//     control that the user wants help information on. Post a private
//     message to the application, then return 1L to indicate that
//     this message was handled.
//
//     When the application receives the private message, it can call
//     WinHelp. WinHelp must NOT be called directly from the
//     FilterFunc routine.
//
//     In this example, post a private PM_CALLHELP message to the
//     dialog box. wParam and lParam can be used to pass context
//     information.
//
//     If the message is not an F1 keystroke, or if nCode is
//     MSGF_MENU, we return 0L to indicate that we did not process
//     this message.
//
//

DWORD FAR PASCAL FilterFunc(nCode, wParam, lParam)
int   nCode;
WORD  wParam;
DWORD lParam;
{
   MSG FAR * ptrMsg;

   if (nCode < 0)                          // MUST return DefHookProc
      return DefHookProc(nCode, wParam, lParam,
```

```
                         (FARPROC FAR *)&lpfnOldHook);

   if (MSGF_DIALOGBOX == nCode)
      {
      ptrMsg = (MSG FAR *)lParam;

      if ((WM_KEYDOWN == ptrMsg->message)
             && (VK_F1 == ptrMsg->wParam))
         {
         // Use PostMessage here to post an application-defined
         // message to the application. Here is one possible call:
         PostMessage(GetParent(ptrMsg->hwnd),
                             PM_CALLHELP, ptrMsg->hwnd, 0L);
         return 1L;                        // Handled it
         }
      else
         return 0L;                        // Did not handle it
      }
   else                                    // I.e., MSGF_MENU
      {
         return 0L;                        // Did not handle it
      }
}
/*--------------------- end FilterFunc  ----------------------------*/
```

Note that FilterFunc returns a DWORD. The documentation in volume 1 of the SDK reference volume 1 that indicates that a filter function should return an integer is incorrect. Note also that not all messages are passed to DefHookProc. The message hook is for this application only, so it is known that no other application needs to see these messages. However, DefHookProc MUST be called when the nCode parameter is less than zero.

4. Export the filter function in the module definition file.

5. Before calling DialogBox, set the hook. After DialogBox returns, unhook the hook. For example, the About Generic dialog box could be called as follows:

```
   case IDM_ABOUT:
     lpProcAbout    = MakeProcInstance(About, hInst);
     lpfnFilterProc = MakeProcInstance(FilterFunc, hInst);
     lpfnOldHook    = SetWindowsHookEx(WH_MSGFILTER,
                                       lpfnFilterProc,
                                       hInst,
                                       NULL);

     DialogBox(hInst, "AboutBox", hWnd, lpProcAbout);

     UnhookWindowsHook(WH_MSGFILTER, lpfnFilterProc);
     FreeProcInstance(lpfnFilterProc);
     FreeProcInstance(lpProcAbout);

     break;
```

6. Have the application respond to the private message by calling

WinHelp.

Additional reference words: 3.00 3.0
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Call the Windows Help Search Dialog Box from Application

Article ID: Q86268

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In the Microsoft Windows version 3.1 environment, an application can invoke the Search dialog box of the Windows Help application independent of the main help window. For example, the Program Manager displays the Search dialog box when the user selects "Search for Help on" from its Help menu.

An application can invoke the Search dialog box via the WinHelp function by specifying HELP_PARTIALKEY as the value for the fuCommand parameter and by specifying a pointer to an empty string for the dwData parameter. The following code demonstrates how to call the Windows Help Search dialog box from an application:

```
LPSTR lpszDummy,
      lpszHelpFile;

// Initialize an empty string
lstrcpy(lpszDummy, "");

// Initialize the help filename
lstrcpy(lpszHelpFile, "c:\\windows\\myhelp.hlp");

// Call WinHelp function
WinHelp(hWnd, lpszHelpFile, HELP_PARTIALKEY, (DWORD)lpszDummy);
```

Additional reference words: SDK 3.10
KBCategory:
KBSubcategory: UsrDlgsRare/misc

PRB: Windows 3.0 Dialog Editor Limitations

Article ID: Q72694

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article lists various limitations of the Windows version 3.0 Dialog Editor.

LIMITATION: Dialog Editor Mishandles Numbers Larger Than 32767

SYMPTOMS
   When a number larger than 32767 is entered into the Dialog Editor, the number is not stored correctly in the DLG file.

CAUSE
   The Dialog Editor uses signed integers for its internal storage.
STATUS
   Microsoft has confirmed that this limitation exists in the Windows version 3.0 Dialog Editor.

LIMITATION: Dialog Box Scroll Bars Positioned Incorrectly

SYMPTOMS
   When the Dialog Editor is used to create a dialog box that has a dialog frame and scroll bars, and the created resource is used in an application, the scroll bars are positioned such that the dialog frame partially overlaps the scroll bars.

RESOLUTION
   Placing scroll bars directly on dialog boxes is neither supported nor recommended.
LIMITATION: A Dialog Box Is Limited to 255 Controls

SYMPTOMS
   After an attempt is made to add the 256th control to a dialog box under construction in the Dialog Editor and the error message box is dismissed, the outline and sizing boxes of the invalid control remain on the screen.

CAUSE
   The Dialog Editor does not handle this error condition properly.
STATUS
   Microsoft has confirmed that this limitation exists in the Windows version 3.0 Dialog Editor.

LIMITATION: #include Files with Comments or Nesting Not Parsed

SYMPTOMS
The Dialog Editor refuses to read a header file and prints an error message.

CAUSE
The error is displayed because the header file contains comments or a nested include file.

RESOLUTION
Remove the comments or nested include from the header file.

LIMITATION: Symbolic Constants Changed to Evaluated Values

SYMPTOMS
The Dialog Editor changes symbolic constants in included header files to their numeric equivalents.

STATUS
Microsoft has confirmed that this limitation exists in the Windows version 3.0 Dialog Editor.

LIMITATION: Long Names of Symbolic Constants Truncated

SYMPTOMS
In the Dialog Editor, the names of long symbols are truncated for display purposes.

STATUS
Microsoft has confirmed that this limitation exists in the Windows version 3.0 Dialog Editor.

LIMITATION: Dialog Menus Not Displayed or Preserved

SYMPTOMS
When a dialog box that has a menu is loaded into the Dialog Editor, the menu is not displayed. When this dialog box is saved from the Dialog Editor, the menu specification is removed.

RESOLUTION
Placing a menu on a dialog box is not recommended or supported.

Additional reference words: 3.0 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Actions Prohibited in System Modal Dialog Boxes

Article ID: Q74332

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The following actions are not supported in a system modal dialog box in the Microsoft Windows graphical environment:

1. Sending an intertask message

2. Changing the focus

3. Activating another application

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrDlgsModal

INF: Dialog Box Frame Styles
Article ID: Q74334

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Dialog boxes can have either the WS_DLGFRAME or the WS_BORDER style. If a dialog box is created with both of these styles, it will have a caption bar instead of the expected frame and border. This is because WS_BORDER | WS_DLGFRAME is equal to WS_CAPTION.

To create a dialog box with a modal dialog frame and a caption, use DS_MODALFRAME combined with WS_CAPTION.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Killing the Parent of a Modal Dialog Box

Article ID: Q74474

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

An application cannot destroy the parent of a modal dialog box. When the modal dialog box ends, control is returned to its parent. If the parent is destroyed, control is given to a non-existent window and the application crashes.

When a parent window receives the destroy message, it is too late to destroy a child modal dialog box. The destroyed dialog box does not return control until after all the existing messages are processed, including the final destruction of the parent window.

Therefore, it is imperative that all generational parents of a modal dialog box be disabled and have no way of receiving a WM_DESTROY message while the modal dialog box is active.

This situation does not affect modeless dialog boxes. Therefore, if the design of a system cannot rule out the above problem, use a modeless dialog box instead of a modal dialog box.

Because the Task Manager can kill any program at any time, there is no absolute protection from the parent being closed. The optimal solution is to forgo modal dialog boxes in favor of modeless dialog boxes. When a modeless dialog box is created, it may disable its parental windows.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsModal

FIX: Painting Problems with DS_SYSMODAL Dialog Boxes

Article ID: Q74506

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9107003

SYMPTOMS
When a system modal dialog box brings up a second system modal dialog box that overlaps it, the portion of the first dialog box hidden by the second dialog box is not repainted when the second dialog box is removed.

STATUS
Microsoft has confirmed this to be a problem in Windows version 3.00. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsModal

INF: Dynamically Changing Icons in a Modal Dialog Box

Article ID: Q74509

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

DLGICONS.ZIP is a code sample in the Software/Data Library that demonstrates how to dynamically change an icon in a modal dialog box. This sample program was created using the tools in the Windows Software Development Kit (SDK) version 3.0.

DLGICONS can be found in the Software/Data Library by searching on the keyword DLGICONS, the Q number of this article, or S13117. DLGICONS was archived using the PKware file-compression utility.

More Information:

The modal dialog box displays an icon by creating a static child window with the SS_ICON style. The lpWindowName parameter to the CreateWindow() function specifies the icon to be displayed. To change the icon, the dialog destroys the child and creates a new child in the same location.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsModal

INF: Using the DS_SETFONT Dialog Box Style

Article ID: Q87344

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In the Microsoft Windows graphical environment, an application can affect the appearance of a dialog box by specifying the DS_SETFONT style bit. DS_SETFONT is available only when the application creates a dialog box dynamically from a memory-resident dialog box template using the CreateDialogIndirect, CreateDialogIndirectParam, DialogBoxIndirect, or DialogBoxIndirectParam function. The second parameter to each of these functions is the handle to a global memory object that contains a DLGTEMPLATE dialog box template data structure. The dwStyle (first) member of the DLGTEMPLATE structure contains style information for the dialog box.

When an application creates a dialog box using one of these functions, Windows determines whether the template contains a FONTINFO data structure by checking for the DS_FONTSTYLE bit in the dwStyle member of the DLGTEMPLATE structure. If this bit is set, Windows creates a font for the dialog box and its controls based on the information in the FONTINFO structure. Otherwise, Windows uses the default system font to calculate the size of the dialog box and the placement and text of its controls.

If Windows creates a font based on the FONTINFO data structure, it sends a WM_SETFONT message to the dialog box. If Windows uses the system default font, it does not send a WM_SETFONT message. A dialog box can change the font of one or more of its controls by creating a font and sending a WM_SETFONT message with the font handle to the appropriate controls.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: WM_PAINTICON Message Removed from Windows SDK Docs

Article ID: Q88192

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The WM_PAINTICON message is not listed in the "Programmer's Reference, Volume 3: Messages, Structures, and Macros" for version 3.1 of the Microsoft Windows Software Development Kit (SDK). The text below explains this decision and provides source code that enables an application to paint its own icon.

More Information:

When an application for the Microsoft Windows operating system registers a window class, it specifies a handle to an appropriate icon in the hIcon member of the WNDCLASS data structure. This can be an application-specific icon loaded from the application's resources or the default application icon provided with Windows.

If the application includes its own icon, Windows sends a WM_PAINTICON message when the user minimizes the application. If the application passes the WM_PAINTICON message to the DefWindowProc function, Windows paints the icon in the appropriate location.

If the application specifies the default icon or no icon, Windows does not send a WM_PAINTICON message when the user minimizes the application; instead, it sends a WM_PAINT message. When the application receives the WM_PAINT message, it can either pass the message to DefWindowProc so that Windows will paint the default icon, or the application can paint the icon dynamically.

Because the application cannot process the WM_PAINTICON message, except to pass it to DefWindowProc, it has been removed from the documentation for version 3.1 of the Windows SDK.

The following code demonstrates how to process the WM_PAINT message to paint the icon. Given a window handle, the IsIconic function indicates whether the window is an icon:

```
   case WM_PAINT:
      {
      PAINTSTRUCT ps;

      BeginPaint(hWnd, &ps);
      if (IsIconic(hWnd))
         {
         SendMessage(hWnd, WM_ICONERASEBKGND, (WORD)ps.hdc, 0);
         DrawIcon(ps.hdc, 0, 0, hIcon);
         }
       EndPaint(hWnd, &ps);
```

```
       }
       break;
```

When the application processes the WM_QUERYDRAGICON message by returning the handle to its icon, Windows converts the icon to a cursor when the user drags the icon window with the mouse.

```
   case WM_QUERYDRAGICON:
      return hIcon;
```

Because the application paints the icon background when it processes the WM_PAINT message, the following code prevents the screen from flashing:

```
   case WM_ERASEBKGND:
      if (IsIconic(hWnd))
         return TRUE;
      else
         return DefWindowProc(...);
      break;
```

DefWindowProc erases the background.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: ControlData Structure Not Completely Documented
Article ID: Q88680

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The ControlData structure documented on page 92 of the "Programmer's Reference, Volume 4: Resources" manual that comes with the Microsoft Windows version 3.1 Software Development Kit (SDK) is not complete. This structure is also not complete in the online Help file (under the heading "Dialog Box Resource").

The correct structure is:

```
struct ControlData {
    WORD  x;
    WORD  y;
    WORD  cx;
    WORD  cy;
    WORD  wID;
    DWORD lStyle;
    union
    {
        BYTE class;     /* if (class & 0x80) */
        char szClass[]; /* otherwise         */
    } ClassID;
    char szText[];
    BYTE cbCreateInfo;  /* added */
    char CreateInfo[];  /* added */
};
```

cbCreateInfo specifies the number of bytes of additional data that follows this item's description and precedes the next item's description (that is, it specifies the length of CreateInfo).

CreateInfo also specifies additional data that the CreateWindow function passes to the WM_CREATE handler of the control (through the lpCreateParams field of the CREATESTRUCT data structure). This field is zero length if cbCreateInfo is zero.

More Information:

When you are using the Windows 3.1 SDK Dialog Editor and Resource Compiler to create dialog box templates that are bound to .EXE files, the cbCreateInfo field is initialized to 0. The only way for an application to use these fields is to create a dialog box template on the fly and call one of the following functions: CreateDialogIndirect, CreateDialogIndirectParam, DialogBoxIndirect, or DialogBoxIndirectParam.

Note that the Windows 3.1 Dialog Manager passes a pointer to the item following the cbCreateInfo field in the dialog template even if cbCreateInfo is 0. This means that controls that rely on the lpCreateParams field in the CREATESTRUCT being NULL when there is no extra creation information will NOT function properly when using a standard Windows dialog resource.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Using DWL_USER to Access Extra Bytes in a Dialog Box

Article ID: Q88358

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Windows extra bytes are used to store private information specific to an instance of a window. For dialog boxes, these extra bytes are already allocated by the dialog manager. The offset to the extra byte location is called DWL_USER.

DWL_USER is the 8th byte offset of the dialog extra bytes. The programmer has 4 bytes (a long) available from this offset for personal use.

CAUTION: DO NOT use more than 4 bytes of these extra bytes, as the rest of them are used by the dialog manager.

Example

```
   DWORD dwNumber = 10;
            .
            .
            .
            .
   case WM_INITDIALOG:
           SetWindowLong(hWnd,DWL_USER,dwNumber); // Store value 10 at
                                                  // byte offset 8
           dwNumber = GetWindowLong(hWnd,DWL_USER); // Retrieve the value
```

Note: GetWindowWord and SetWindowWord could be used instead.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Do Not Call Message Boxes Via Radio Buttons

Article ID: Q62741

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When programming calls to MessageBox() in response to dialog box controls, you may observe the following phenomenon.

An application-modal message box that is opened within the processing logic for a radio button control crashes the system (you keep getting message boxes until USER runs out of local heap space). There is no problem as soon as you replace the radio button style by a push button style.

Although this is a problem in Windows, there should not really be a need to use this feature at all. Radio buttons are designed to be a mechanism to choose between mutually exclusive options (for more information, refer to the CUA style guide). Opening a message box as a direct response to the selection of a radio button is not a preferred programming technique. If the selected option raises some kind of error, the radio button can always be disabled using the EnableWindow() function.

Additional reference words: 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: FatalExit 0x0001 Possible If WM_CTLCOLOR Used Improperly
Article ID: Q42458

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:
An application can change the colors of a dialog box by creating a brush of the desired color and processing the WM_CTLCOLOR message to return a handle to the brush. If the brush is not a stock object, the application should create the brush before the dialog box is displayed and store its handle in a global or static variable. Do not use the following code in a dialog box procedure:

```
   case WM_CTLCOLOR:  // Create green edit fields
      if (HIWORD(lParam) == CTLCOLOR_EDIT)
         return CreateSolidBrush(RGB(0, 255, 0));
      break;
```

Windows sends a WM_CTLCOLOR message to the dialog box procedure each time any changes are made to the dialog box (such as when the user presses a button or moves the focus to a new control). In the code above, each time the application processes a WM_CTLCOLOR message for an edit control, the application creates a new brush which is stored in the GDI data segment. If the GDI data segment fills completely, Windows generates a FatalExit 0x0001 (insufficient memory for allocation) error for each subsequent allocation.

More Information:

To avoid this error, modify the application to call the CreateSolidBrush function once, either outside the application's message loop or during the processing of the WM_CREATE message. Store the returned brush handle in a global or static variable. An advantage of this method is that the brush can be used in any part of the application without consuming any additional system resources.

The following code provides a skeleton for the WinMain procedure and for the dialog procedure to implement this method:

```
// WinMain
HBRUSH  hBrushGreen; // Global variable

WinMain (...)
{
   // Register window class and create window

   hBrushGreen = CreateSolidBrush(RGB(0, 255, 0));

   while (GetMessage(...))
     {
     TranslateMessage(...);
```

```
      DispatchMessage(...);
      }

   DeleteObject(hBrushGreen);
   return msg.wparam;
}

// Dialog box procedure
   case WM_CTLCOLOR: // Green edit fields
      if (HIWORD(lParam) == CTLCOLOR_EDIT)
         return hBrushGreen;
      break;
```

Additional reference words: 1.x 2.03 2.10 3.00 3.10 2.x
KBCategory:
KBSubcategory: UsrDlgsIds

INF: Centering a Dialog Box on the Screen

Article ID: Q74798

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

When an application developed for the Microsoft Windows graphical environment displays a dialog box, centering the dialog box on the screen is sometimes desirable. However, on systems with high-resolution displays, the application displaying the dialog box may be nowhere near the center of the screen. In these cases, it is preferable to place the dialog near the application requesting input.

To center a dialog box on the screen before it is visible, add the following lines to the processing of the WM_INITDIALOG message:

```
{
RECT rc;

GetWindowRect(hDlg, &rc);

SetWindowPos(hDlg, NULL,
   ((GetSystemMetrics(SM_CXSCREEN) - (rc.right - rc.left)) / 2),
   ((GetSystemMetrics(SM_CYSCREEN) - (rc.bottom - rc.top)) / 2),
   0, 0, SWP_NOSIZE | SWP_NOACTIVATE);
}
```

This code centers the dialog horizontally and vertically.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Specifying Time to Display and Remove a Dialog Box
Article ID: Q74888

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

It is possible to modify the timing of the display of a dialog box. For example, an application has its copyright message in a dialog box that does not have any push buttons. This dialog is designed to be displayed for five seconds and then to disappear. This article discusses a method to implement this functionality.

More Information:

Windows draws the dialog box on the screen during the processing of a WM_PAINT message. Because all other messages (except for WM_TIMER messages) are processed before WM_PAINT messages, there may be some delay before the dialog box is painted. This may be avoided by placing the following code in the processing of the WM_INITDIALOG message:

```
   ShowWindow(hDlg);
   UpdateWindow(hDlg);
```

This code causes Windows to send a WM_PAINT message to the dialog box, bypassing the other messages that may be in the application's queue.

To keep the dialog box on the screen for a particular period of time, a timer should be created during the processing of the WM_INITDIALOG message. When the WM_TIMER message is received, call EndDialog() to close the dialog box.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Changing the Font Used by Dialog Controls in Windows

Article ID: Q74737

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In Windows 3.0, there are two ways to specify the font used by dialog controls:

1. The FONT statement can be used in the dialog template to specify
   the font used by ALL the controls in the dialog box.

   -or-

2. The WM_SETFONT message can be sent to one or more dialog controls
   during the processing of the WM_INITDIALOG message.

If a font is specified in the dialog template, the controls will use a bold version of that font. The following code demonstrates how to change the font used by dialog box controls to a nonbold font using WM_SETFONT. The font should be deleted with DeleteObject() when the dialog box is closed.

```
HWND hDlg;
HFONT hDlgFont;
LOGFONT lFont;

case WM_INITDIALOG:
     /* Get dialog font and create non-bold version */
     hDlgFont = NULL;
     if ((hDlgFont = (HFONT)SendMessage(hDlg, WM_GETFONT, 0, 0L))
              != NULL)
        {
        if (GetObject(hDlgFont, sizeof(LOGFONT), (LPSTR)&lFont)
                 != NULL)
           {
           lFont.lfWeight = FW_NORMAL;
           if ((hDlgFont = CreateFontIndirect(&lFont)) != NULL)
              {
              SendDlgItemMessage(hDlg, CTR1, WM_SETFONT, hDlgFont, 0L);
              // Send WM_SETFONT message to desired controls
              }
           }
        }
     else  // user did not specify a font in the dialog template
        {  // must simulate system font
        lFont.lfHeight = 13;
        lFont.lfWidth = 0;
        lFont.lfEscapement = 0;
        lFont.lfOrientation = 0;
        lFont.lfWeight = 200; // non-bold font weight
```

```
        lFont.lfItalic = 0;
        lFont.lfUnderline = 0;
        lFont.lfStrikeOut = 0;
        lFont.lfCharSet = ANSI_CHARSET;
        lFont.lfOutPrecision = OUT_DEFAULT_PRECIS;
        lFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
        lFont.lfQuality = DEFAULT_QUALITY;
        lFont.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
        lFont.lfFaceName = (LPSTR)NULL;
        hDlgFont = CreateFontIndirect(&lFont);

        SendDlgItemMessage(hDlg, CTR1, WM_SETFONT, hDlgFont,
           (DWORD)TRUE);
        // Send WM_SETFONT message to desired controls
        }

    return TRUE;
    break;
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

INF: Creating a Progress Dialog with a Cancel Option

Article ID: Q76415

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following article describes the mechanisms necessary to implement a progress or activity indicator with a "Cancel Operation" option, to be used for lengthy and CPU-intensive subtasks of an application. Examples of operations using this include: copying multiple files, directory searches, or printing large files.

More Information:

The progress dialog is implemented in two steps:

1. Initialize the dialog before starting lengthy or CPU intensive subtask.

2. After each unit of the subtask is complete, call ProgressYield() to determine if the user has canceled the operation and to update the progress or activity indicator.

This is the description of the progress dialog procedure. The procedure uses a global variable (Cancel) to inform the CPU-intensive subtask that the user has indicated a desire to terminate the subtask.

```
WORD Cancel = FALSE;     /* This must be global to all modules */
                         /* which call ProgressYield()         */

BOOL FAR PASCAL ProgressDlgProc(hDlg, message, wParam, lParam)
HWND hDlg;
unsigned message;
WORD wParam;
DWORD lParam;
{
   switch (message)
      {
      .
      /* use other messages to update the progress or activity */
      /* indicator                                             */
      .
      case WM_COMMAND:
         switch (wParam)
            {
            case ID_CANCEL:    /* ID_CANCEL = 2 */
               Cancel = TRUE;

            default:
               return FALSE;
            }
```

```
      .
      .
      default:
         return FALSE;
      }
}
```

The following describes the ProgressYield procedure, which should be called after each unit of the CPU intensive subtask is completed. The ProgressYield procedure uses the IsDialogMessage function (described in the "Microsoft Windows Software Development Kit Reference Volume 1"). IsDialogMessage will convert keyboard messages into selection commands for the corresponding dialog box as outlined in Table 1.7 beginning on page 1-51.

```
void ProgressYield(HWND hwnd)
{
   MSG   msg;

   /* remove all available messages for any window that belong */
   /* to the current application                               */
   while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
      /* Translate and Dispatch the given message if the window */
      /* handle is null or the given message is not for the    */
      /* modeless dialog box hwnd.                             */
      if (!hwnd || !IsDialogMessage(hwnd, &msg))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
      }
}
```

The following describes how to incorporate the progress dialog as part of an application's subtask that is CPU-intensive. The PROGRESS_DLG resource should contain a button with an ID of 2, since this is the wParam of the WM_COMMAND that will be sent when the user presses the ESC key. The button should also have the BS_DEFPUSHBUTTON style so that the ENTER key will also result in the termination of the CPU-intensive subtask.

```
   FARPROC lpProgressProc;
   HWND    hwndProgress;            /* This needs to be global if */
                                    /* accessed by other modules  */
   .
   .
   /* Initialize before starting CPU intensive work              */

   lpProgressProc = MakeProcInstance(ProgressDlgProc,
                                     hInst);  /* current instance */

   hwndProgress = CreateDialog(hInst,         /* current instance */
                              "PROGRESS_DLG", /* resource         */
                              hwndParent,     /* Parent handle    */
```

```
                              lpProgressProc);/* instance address */
   ShowWindow(hwndProgress);
   .
   .
   /* Start CPU intensive work here */
      .
      .
      /* Before or after each unit of work, the application */
      /* should do the following:                           */
      ProgressYield(hwndProgress);
      if (Cancel == TRUE)
         break;   /* Terminate CPU-intensive work immediately */
      .
      .
   /* End CPU-intensive work here */
   .
   .
   DestroyWindow(hwndProgress);
   FreeProcInstance(lpProgressProc);
   .
   .
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDlgsRare/misc

PRB: Infinite Loop When Maneuvering Through Dialog Box Control
Article ID: Q92905

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

An application enters an infinite loop when you use the TAB or arrow keys to navigate through dialog box controls. The problem occurs only when the dialog box has another dialog box as a child. When you use the TAB key, the problem occurs only when the child dialog box has the WS_MAXIMIZEBOX or WS_TABSTOPS style.

This article describes the problem in detail together with two possible solutions. A sample is also provided that demonstrates the problem and one of the workarounds.

SYMPTOMS

Problem 1: When arrow keys are used to maneuver through a dialog box's last control (the last control in the parent dialog box), the focus is passed to the next child (in this case, the child dialog box) and the application goes into an infinite loop.

Problem 2: When the TAB key is used to maneuver through the dialog box's last control, the focus is then passed to the next child (again the child dialog box) and the application goes into an infinite loop.

The infinite loop is clearly visible when spying on the child dialog box window. The child dialog box window continually receives WM_GETDLGCODE messages.

CAUSE

The dialog box manager confuses the child dialog box window with a child control.

When maneuvering through controls in a dialog box, the dialog box manager looks through its children and checks for the next child to pass focus to. In order to do that, the dialog box manager sets the focus to the child control and generates messages to complete the process.

Using the Arrow Keys to Navigate Through Controls

WS_GROUP defines the start of a group within a dialog box. All controls defined between two WS_GROUP styles are members of a group. Arrow keys can then be used to navigate between members of a group.

Normally a dialog box does not have WS_GROUP for a style; therefore, the child dialog box does not have a WS_GROUP but a previous control does have one. Based on the definition of a group, the child dialog box is considered a member of the last group by the dialog box manager. When

the arrow key is used on the last control, the dialog box manager checks for the next child control in that group to pass the focus to, in this case the child dialog box window. The dialog box manager then tries to set focus to the window and generates necessary messages to complete the process. These generated messages are meaningful to a control but not to a window. One of these generated messages is WM_GETDLGCODE, which causes the infinite loop.

Using the TAB Key to Navigate Through Controls

WS_TABSTOPS defines which control has a tab stop. When you tab through controls, the dialog box manager looks for the next child that has the WS_TABSTOPS style and changes the focus to that child.

Normally a dialog box does not use WS_TABSTOPS for a style. It is quite normal, however, to use WS_MAXIMIZEBOX for a dialog box style. The child dialog box in this case has WS_MAXIMIZEBOX. The values for WS_TABSTOPS and WS_MAXIMIZEBOX are the same in WINDOWS.H. Because the dialog box manager expects a child control and not another window, the WS_MAXIMIZEBOX is interpreted as WS_TABSTOPS. So, when you tab on the last control of the parent dialog box, the dialog box manager tries to set the focus to the next child with WS_TABSTOPS, in this case the child dialog box. Then the dialog box manager generates the necessary messages to complete the process. These generated messages are meaningful to a control but not to a window. One of these messages is WM_GETDLGCODE, which causes the infinite loop.

RESOLUTION

Workarounds are described in the More Information section below.

More Information:

- Do not include the WS_MAXIMIZEBOX style for the child dialog box to resolve the TAB key problem, and add WS_GROUP to the child dialog box to resolve the arrow key problem.

- Subclass the last control and handle the change of focus in the dialog box. This method is outlined below:

  1. Subclass the last control in the parent dialog box. For more information on subclassing a control, query on the following words in the Microsoft Knowledge Base:

     safe and subclassing
  2. In the subclass procedure for the control, process the WM_GETDLGCODE message by:

     a. Passing the WM_GETDLGCODE to the button window procedure.
     b. Or'ing the return value of (a.) with DLGC_WANTMESSAGE and returning it.

     The following is sample code demonstrating the WM_GETDLGCODE message:

        lpButtonProcOrg is a pointer to the button window procedure. This is a global and was set before setting the pointer to the subclass procedure for the button.

To do this call:

```
        lpButtonProgOrg = (FARPROC)
        GetWindowLong(GetDlgItem(hCancelBtn,GWL_WNDPROC);
        :
        :

  case WM_GETDLGCODE:
    lRet = CallWindowProc(lpButtonProcOrg,hWnd,iMessage,
                          wParam,lParam);
    if (lParam)
      {
      lpmsg= (LPMSG) lParam;
      if (lpmsg->message == WM_KEYDOWN)
       {
       if ( (lpmsg->wParam == VK_TAB)   ||
            (lpmsg->wParam == VK_DOWN)  ||
            (lpmsg->wParam == VK_UP)    ||
            (lpmsg->wParam == VK_RIGHT) ||
            (lpmsg->wParam == VK_LEFT) )
         lRet |= DLGC_WANTMESSAGE;
       }
      }
    return (lRet);
```

For more information on WM_GETDLGCODE, query on the following word in the Microsoft Knowledge Base:

WM_GETDLGCODE

3. In the subclass procedure, process the WM_KEYDOWN message by:

   a. Set the focus to the first control.
   b. If the first control is not a push button, then the process is complete; otherwise, the dark border needs to be removed from whichever button that has it, and placed on the first button, as follows:

      1) First send a WM_GETDLGCODE.

      2) If the return value is DLGC_DEFPUSHBUTTON, then send a BM_SETSTYLE message to the button that has the dark border, with BS_PUSHBUTTON for wParam. This will cause the dark border to be removed.

      3) Now the dark border needs to be placed on the button that has the focus (that is, the first button). To do this, send BM_SETSTYLE to the first button, with BS_DEFPUSHBUTTON for wParam.

Sample Code Demonstrating the WM_KEYDOWN Message

```
case WM_KEYDOWN:

   // Check for keys that need this processing such as the
   // TAB and arrow keys.
   if ( (wParam == VK_TAB)   ||
```

```
            (wParam == VK_DOWN)  || (wParam == VK_UP) ||
            (wParam == VK_RIGHT) || (wParam == VK_LEFT) )
          {
          // Set the focus to the first control, Button #1
          // in this case.
          hwndBtn1 = GetDlgItem(hParentDlg,IDBUTTON1);
          SetFocus(hwndBtn1);

          // Loop through all the controls and remove the
          // dark border that the previous default push
          // button has.
          hwndCtrls = GetWindow(hParentDlg, GW_CHILD);
          while (hwndCtrls)
              {
              wRet = (WORD)(DWORD)SendMessage(hwndCtrls,
                                   WM_GETDLGCODE, 0, 0L);

             if (wRet & DLGC_DEFPUSHBUTTON)
               SendMessage(hwndCtrls, BM_SETSTYLE,
                          (WPARAM)BS_PUSHBUTTON, TRUE);

             hwndCtrls = GetWindow(hwndCtrls, GW_HWNDNEXT);
             }

         // Give the hwndBtn1 button the default push button
         // border.
          SendMessage(hwndBtn1,  BM_SETSTYLE,
                      (WPARAM)BS_DEFPUSHBUTTON, TRUE);

          return(0L);
          }

       break;
```

DLGTAB is a file in the Software/Data Library that demonstrates both the TAB and ARROW key problems, together with the second method of resolving the problem, which is listed above.

DLGTAB can be found in the Software/Data Library by searching on the word DLGTAB, the Q number of this article, or S13724. DLGTAB was archived using the PKware file-compression utility.

To get the dialog boxes, from the File menu, choose Dialog Tab. Two dialog boxes will be displayed: one is the Parent dialog box and the other is the Child dialog box.

There are two option buttons in the Parent dialog box: one demonstrates the problem and the other the solution. When the Demo Problem option button is selected, the sample demonstrates the problem. When the Fix Problem option button is selected, the sample demonstrates the workaround. The default button is the workaround for the problem.

Additional reference words: 3.1 3.10
KBCategory:
KBSubcategory: UsrDlgsTabbing

SAMPLE: Changing Background and Text Color of Message Box

Article ID: Q99808

Summary:

MSGCOLOR is a file in the Software/Data Library that demonstrates how to change the background and text color of a message box by subclassing the message box.

MSGCOLOR can be found in the Software/Data Library by searching on the word MSGCOLOR, the Q number of this article, or S14186. MSGCOLOR was archived using the PKware file-compression utility.

More Information:

The MSGCOLOR sample implements the ColorMessageBox() function, which creates a message box whose background and text color can be changed. The function subclasses the message box and processes WM_CTLCOLOR to modify the colors. Subclassing the message box requires obtaining the window handle of the message box. The window handle is obtained using a WH_CBT hook. The hook is called when the message box is created, and provides access to the window handle.

Windows 3.1 is required for the sample.

Additional reference words: 3.10 softlib MSGCOLOR.ZIP
KBCategory:
KBSubcategory: UsrDlgsMsgbox

INF: Using ENTER Key from Edit Controls in a Dialog Box

Article ID: Q102589

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) version 3.1

SUMMARY
=======

Windows applications often display data-entry dialog boxes to request information from users. These dialog boxes may contain several edit controls and two command (push) buttons, labeled OK and CANCEL. An example of a data-entry dialog box is one that requests personal information, such as social security number, address, identification (ID) number, date/time, and so on, from users. Each of these items is entered into an edit control.

By default, the TAB key is used in a dialog box to shift focus between edit controls. As a common user-interface, however, one could also use the ENTER (RETURN) key to move between the edit controls (for example, after the user enters a piece of information, pressing ENTER moves the focus to the next field).

There are two ways of enabling the use of the ENTER key to move between edit controls. One method is to make use of WM_COMMAND and the notification messages that come with it in the dialog box for edit controls and buttons. Another method involves subclassing the edit controls.

MORE INFORMATION
================

Method I: (WM_COMMAND)

This method is based on the following behavior of dialog boxes (Dialog Manger) and focus handling in Windows.

If a dialog box or one of its controls currently has the input focus, then pressing the ENTER key causes Windows to send a WM_COMMAND message with the idItem (wParam) parameter set to the ID of the default command button. If the dialog box does not have a default command button, then the idItem parameter is set to IDOK by default.

When an application receives the WM_COMMAND message with idItem set to the ID of the default command button, the focus remains with the control that had the focus before the ENTER key was pressed. Calling GetFocus() at this point returns the handle of the control that had the focus before the ENTER key was pressed. The application can check this control handle and determine whether it belongs to any of the edit controls in the dialog box. If it does, then the user was entering data into one of the edit controls and after doing so,

pressed ENTER. At this point, the application can send the WM_NEXTDLGCTL message to the dialog box to move the focus to the next control.

However, if the focus was with one of the command buttons (CANCEL or OK), then GetFocus() returns a button control handle, at which point one can dismiss the dialog box. The pseudo code for this logic resembles the following in the application's dialog box procedure:

```
     case WM_COMMAND:

     if(wParam=IDOFDEFBUTTON || IDOK) {
          // User has hit the ENTER key.

          hwndTest = GetFocus() ;
          retVal = TesthWnd(hWndTest) ;

          //Where retVal is a boolean variable that indicates whether
        //the hwndTest is the handle of one of the edit controls.

          if(hwndTest) {
         //Focus is with an edit control, so do not close the dialog.
            //Move focus to the next control in the dialog.

         PostMessage(hDlg, WM_NEXTDLGCTL, 0, 0L) ;
         return TRUE ;
          }
          else {
            //Focus is with the default button, so close the dialog.
            EndDialog(hDlg, TRUE) ;
            return FALSE ;
          }
        }
     break ;
```

Method II

This method involves subclassing/superclassing the edit control in the dialog box. Once the edit controls are subclassed or superclassed, all keyboard input is sent the subclass/superclass procedure of the edit control that currently has input focus, regardless of whether or not the dialog box has a default command button. The application can trap the key down (or char) messages, look for the ENTER key, and do the processing accordingly. The following is a sample subclass procedure that looks for the ENTER key:

```
//*---------------------------------------------------------------------
//| Title:
//|     SubClassProc
//|
//| Parameters:
//|     hWnd           - Handle to the message's destination window
//|     wMessage       - Message number of the current message
//|     wParam         - Additional info associated with the message
//|     lParam         - Additional info associated with the message
//|
```

```
//| Purpose:
//|     This is the window procedure used to subclass the edit control.
//*-----------------------------------------------------------------------

long FAR PASCAL SubProc(HWND hWnd, WORD wMessage,WORD wParam,LONG lParam)
{

    switch (wMessage)
     {

          case WM_GETDLGCODE:
            return (DLGC_WANTALLKEYS |
                    CallWindowProc(lpOldProc, hWnd, wMessage,
                                   wParam, lParam));

          case WM_CHAR:
          //Process this message to avoid message beeps.
         if ((wParam == VK_RETURN) || (wParam == VK_TAB))
           return 0;
            else
           return (CallWindowProc(lpOldProc, hWnd,
                                     wMessage, wParam, lParam));


       case WM_KEYDOWN:
            if ((wParam == VK_RETURN) || (wParam == VK_TAB)) {
              PostMessage (ghDlg, WM_NEXTDLGCTL, 0, 0L);
              return FALSE;
            }

         return (CallWindowProc(lpOldProc, hWnd, wMessage,
                                   wParam, lParam));
         break ;

       default:
            break;

     } /* end switch */
```

Additional reference words: 3.10 push RETURN keydown
KBCategory:
KBSubcategory: UsrDlgsTabbing

PRB: Min/Max Boxes Do Not Work with DS_MODALFRAME
Article ID: Q102645

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows,
   versions 3.0 and 3.1

SYMPTOMS
========

When I use the WS_MIMIMIZEBOX or WS_MAXIMIZEBOX styles in a dialog box with a DS_MODALFRAME style, the up/down arrows are offset up and to the right when I click them with the mouse.

CAUSE
=====

The WS_MINIMIZEBOX and WS_MAXIMIZEBOX styles cannot be used with the DS_MODALFRAME frame style. (Menus are also incompatible with this frame style. For more information, query on the following words in the Microsoft Knowledge Base:

   ds_modalframe and ws_sysmenu

RESOLUTION
==========

Use a different frame style, such as WS_BORDER.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDlgsMsgBox

INF: Implementing the Drag-Drop Protocol
Article ID: Q83543

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Windows 3.1 supports four new functions and one new message that enable an application to implement the drag-drop protocol. DRAGDROP is a file in the Software/Data Library that demonstrates using all four functions and the message. When the user drags and drops a file onto DRAGDROP, the name of the file is added to a list box in DRAGDROP.

DRAGDROP can be found in the Software/Data Library by searching on the word DRAGDROP, the Q number of this article, or S13387. DRAGDROP was archived using the PKware file-compression utility.

More Information:

An application that can accept files dragged from the File Manager calls the DragAcceptFiles function specifying one or more of its windows. When the user drags a file from the File Manager into the window specified in the DragAcceptFiles call, and releases the left mouse button, File Manager sends the application a WM_DROPFILES message. (File Manager sends a WM_DROPFILES message only to applications that have registered a window with DragAcceptFiles.)

One of the parameters to WM_DROPFILES contains a handle to an internal data structure. The DragQueryFile function retrieves a number of dropped files and their names from the data structure and returns this information to the application. Likewise, DragQueryPoint retrieves the position of the mouse cursor when the file(s) was dropped.

To release the memory allocated by Windows for the WM_DROPFILES data structure, the application must call the DragFinish function after it retrieves the applicable data.

Additional reference words: 3.10 softlib DRAGDROP.ZIP
KBCategory:
KBSubcategory: UsrDnd

INF: Windows Version 3.00 Drag and Drop Protocol

Article ID: Q65819

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Certain operations can be performed by direct manipulation of icons in the Windows version 3.0 File Manager and Program Manager. One of the actions that is allowed is the ability to move a program among Program Manager groups by dragging its icon from one group and dropping it into another group.

There are messages in Windows that are currently undocumented that support this behavior. These messages form the "Drag and Drop Protocol."

Development work is still progressing on this protocol, which will be changing in a future version of Windows. Once the necessary work is complete, this protocol will be thoroughly documented and application developers will be encouraged to use it in their applications where ever it is appropriate.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrDndRare/misc

INF: Microsoft Drag-Drop Server Strategy

Article ID: Q81047

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Windows version 3.1 supports a drag-drop protocol that enables an application to be the client of File Manager. When an application implements the drag-drop feature, a user can select one or more files in File Manager, drag them to a client application, and drop them. At this point, the client application receives a message and it can determine which files were dropped and the exact location at which the files were dropped.

At present, version 3.1 of the File Manager is the only server application that supports the drag-drop protocol. Although no Windows function is available to enable an application to become a drag-drop server, Microsoft intends to provide this functionality in a future release of Windows.

The drag-drop interface mechanism will change to provide additional functionality that can be used by a future server application to support application cooperation and shell integration. The new mechanism will continue to support the Windows 3.1 drag-drop client protocol.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrDnd

INF: Using Drag-Drop in an Edit Control or a Combo Box

Article ID: Q86724

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In the Microsoft Windows environment, an application can register an edit control or a combo box as a drag-drop client through the DragAcceptFiles function. The application must also subclass the control to process the WM_DROPFILES message that Windows sends when the user drops a file.

More Information:

The following seven steps demonstrate how to implement drag-drop in an edit control. The procedure to implement drag-drop in a combo box is identical.

1. Add SHELL.LIB to the list of libraries required to build the file.

2. Add the name of the subclass procedure (MyDragDropProc) to the EXPORTS section of the module definition (DEF) file.

3. Include the SHELLAPI.H file in the application's source code.

4. Declare the following procedure and variables:

```
    BOOL FAR PASCAL MyDragDropProc(HWND, unsigned, WORD, LONG);

    FARPROC lpfnDragDropProc, lpfnOldEditProc;
    char    szTemp64[64];
```

5. Add the following code to the initialization of the dialog box:

```
    case WM_INITDIALOG:
       // ... other code

       // ------- edit control section --------
       hWndTemp = GetDlgItem(hDlg, IDD_EDITCONTROL);
       DragAcceptFiles(hWndTemp, TRUE);

       // subclass the drag-drop edit control
       lpfnDragDropProc = MakeProcInstance(MyDragDropProc, hInst);

       if (lpfnDragDropProc)
          lpfnOldEditProc = SetWindowLong(hWndTemp, GWL_WNDPROC,
                 (DWORD)(FARPROC)lpfnDragDropProc);
       break;
```

6. Write a subclass window procedure for the edit control.

```
BOOL FAR PASCAL MyDragDropProc(HWND hWnd, unsigned message,
                               WORD wParam, LONG lParam)
{
   int wFilesDropped;

   switch (message)
      {
   case WM_DROPFILES:
      // Retrieve number of files dropped
      // To retrieve all files, set iFile parameter
      // to -1 instead of 0
      wFilesDropped = DragQueryFile((HDROP)wParam, 0,
            (LPSTR)szTemp64, 63);

      if (wFilesDropped)
         {
         // Parse the file path here, if desired
         SendMessage(hWnd, WM_SETTEXT, 0, (LPSTR)szTemp64);
         }
      else
         MessageBeep(0);

      DragFinish((HDROP)wParam);
      break;

   default:
      return CallWindowProc(lpfnOldEditProc, hWnd, message,
            wParam, lParam);
      break;
   }
   return TRUE;
}
```

7. After the completion of the dialog box procedure, free the edit control subclass procedure.

```
if (lpfnDragDropProc)
   FreeProcInstance(lpfnDragDropProc);
```

Additional reference words: 3.10 combobox
KBCategory:
KBSubcategory: UsrExtShell

SAMPLE: Connect Net Drive--a File Manager Extension

Article ID: Q99862

Summary:

The NETCON.DLL program sample demonstrates how to create a dynamic-link library (DLL) that will extend the File Manager application in Microsoft Windows versions 3.1 and later. The program sample exports a function called FMExtensionProc(), which File Manager sends messages to during its initialization. NETCON adds a Quick Connect menu to File Manager (in Windows for Workgroups, NETCON adds a toolbar button and a help string to be displayed in the status bar when the toolbar button is selected). When the Quick Connect menu item is selected, NECON.DLL brings up a Connect Network Drive dialog box. The program uses the following network application programming interface (API), which must be supported by the network that Windows is running on:

- WNetAddConnection()
- WNetGetConnection()

NETCON.DLL also maintains the MRU_Files section of WIN.INI in the exact manner that Windows for Workgroups does. Unlike the Connect Network Drive dialog box that comes with Windows for Workgroups, this dialog box does not browse for available drives during initialization, and therefore startup time may be significantly faster.

NETCON can be found in the Software/Data Library by searching on the word NETCON, the Q number of this article, or S14183. NETCON was archived using the PKware file-compression utility.

More Information:

To enable File Manger to use the DLL, and the following line to the [AddOns] section of WINFILE.INI:

```
Quick Net Connection = netcon.dll
```

Additional reference words: 3.1 3.10
KBCategory:
KBSubcategory: UsrExtFileman

SAMPLE: Accelerators for File Manager Extensions

Article ID: Q100357

Summary:

File Manager extensions cannot directly use accelerators because extensions do not have their own message pump. File Manager pulls messages on behalf of the extensions, and does not load and translate extensions's accelerators.

XTENACCL is a sample in the Software Library that provides a workaround to this limitation. XTENACCL can be found in the Software/Data Library by searching on the word XTENACCL, the Q number of this article, or S14193. XTENACCL was archived using the PKware file-compression utility.

More Information:

XTENACCL installs a task-specific message hook (WH_GETMESSAGE) and calls TranslateAccelerator() on every WM_KEY and WM_KEYDOWN message. If an appropriate accelerator is found, TranslateAccelerator() passes a WM_COMMAND to the main window.

The main window of File Manager handles and processes the WM_COMMAND message passed to it by TranslateAccelerator(). However, File Manager does not know how to handle that message because wParam (containing menu IDs) is defined by the extension. To interpret the correct value, File Manager dynamically changes an extension's menu IDs to suit its own needs. Therefore, XTENACCL traps the WM_COMMAND message passed by TranslateAccelerator() and routes it to a child window of its own. The child window in turn processes the menu command.

To intercept the WM_COMMAND message, XTENACCL is using a second message hook that is of type WH_CALLWNDPROC. Because TranslateAccelerator() sends a WM_COMMAND instead of posting it, this second hook was needed.

Additional reference words: 3.1 3.10 file manager extension hooks
KBCategory:
KBSubcategory: UsrExtFileman

INF: Explanation of the NEWCPLINFO Structure
Article ID: Q103315

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1

The following is an explanation of the NEWCPLINFO structure:

dwSize: Specifies the length of the structure, in bytes. Set this to sizeof(NEWCPLINFO).

dwFlags: Specifies Control Panel flags. This is field is currently unused. Set this field to NULL.

dwHelpContext: Specifies the context number for the topic in the help project (.HPJ) file that displays when the user selects help for the application. If dwData is non-NULL, Windows Help will be invoked with the HELP_CONTEXT fuCommand with dwHelpContext as dwData. If dwHelpContext is NULL, Windows Help is invoked with the HELP_INDEX fuCommand.

lData: Specifies data defined by the application. This is passed back to the application in the CPL_DBLCLK, CPL_SELECT, and CPL_STOP messages via lParam2.

hIcon: Identifies an icon resource for the application icon. This icon is displayed in the Control Panel window.

szName: Specifies a null-terminated string that contains the application name. The name is the short string displayed below the application icon in the Control Panel window. The name is also displayed in the Settings menu of Control Panel.

szInfo: Specifies a null-terminated string containing the application description. The description is displayed at the bottom of the Control Panel window when the application icon is selected.

szHelpFile: Specifies a null-terminated string that contains the path of the help file, if any, for the application. If this field is unused, set it to NULL. The Control Panel will invoke a default Windows Help file when this field is NULL.

Additional reference words: 3.10 cpl control panel extension
KBCategory:
KBSubcategory: UsrExtCtlpanel

INF: Windows Cardfile File Format

Article ID: Q79192

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article documents the file format used by Windows Cardfile. All numbers in this document, including those in the descriptive text, should be interpreted as hexadecimal numerals. All data pointers and count bytes in the file are unsigned binary/hexadecimal integers in least-to-most significant format. All text in the file is saved in low ASCII format. In the text section of a card's data, <CR> is always followed by <LF>.

More Information:

The Cardfile file format is as follows:

Byte #	Description
0 - 2	Signature bytes -- always "MGC" (4D 47 43)
3 - 4	Number of cards in file

Beyond the first 5 bytes are the index lines -- the information about the top line of each card. The first index entry begins at byte 5 in the file, and successive entries begin 34 bytes after the beginning of the last index entry (the second entry at byte 39, the third entry at byte 6D, and so forth). The format for each index line entry is as follows:

0 - 5	Null bytes, reserved for future use (should all be 00)
6 - 9	Absolute position of card data in file
A	Flag byte (00)
B - 32	Index line text
33	Null byte; indicates end of index entry

After the last index entry, each card's data is stored. Card data will be in one of four general formats: graphic and text, text only, graphic only, and blank. Blank cards consist of 4 null bytes; the other card formats are below:

Graphic & Text	Text Only	Graphic Only	
0 - 1	0 - 1#	0 - 1	Length of graphic bitmap #
2 - 3	*	2 - 3	Width of graphic *
4 - 5	*	4 - 5	Height of graphic *
6 - 7	*	6 - 7	X-coordinate of graphic *
8 - 9	*	8 - 9	Y-coordinate of graphic *

```
   A - x          *         A - x       Bitmap of graphic *
   x+1 - x+2    2 - 3     x+1 - x+2#    Length of text entry #
   x+3 - y      4 - z*                  Text *

   x = 9 + length of bitmap
   y = x + 2 + length of text entry
   z = 3 + length of text entry
   # - These bytes are null if no bitmap/text
   * - These bytes do not exist if no bitmap/text
```

The first byte of any card's data entry is pointed to by bytes 6-9 in the index entry. Note that no null byte is used to indicate the end of the card's data entry; the next card's data entry immediately follows the last byte of the previous entry, which will be null only if the previous card has no text (null length of text entry).

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrFmtRare/misc

INF: Windows Program Manager Edits GRP Files

Article ID: Q65879

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the "Microsoft Windows Software Development Kit Guide to Programming" version 3.0, Chapter 22 describes Dynamic Data Exchange (DDE), the means by which two or more applications can communicate in the Windows environment.

The Windows version 3.0 Program Manager can be directed to create and maintain Program Manager groups and their accompanying GRP files. Section 22.4.4 of the "Microsoft Windows Software Development Kit Guide to Programming" version 3.0 describes the Program Manager DDE command set. Commands are available that create, display, and delete groups, add items to groups, and exit the Program Manager.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrFmtGrp

INF: Windows Paintbrush File Format
Article ID: Q79342

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

This article provides the format of Windows Paintbrush files.

More Information:

Paintbrush files are preceded by 32 bytes of header information. Following the header is an index table into run-length encoded data.

Header

The header structure is defined in the C programming language, as follows:

```
typedef struct
{
  WORD  key1;
  WORD  key2;
  WORD  dxFile;
  WORD  dyFile;

  /* Note: PBRUSH.EXE version 3.0 and 3.1 ignore the remaining values */
  /* in this structure. Future versions may or may not.               */

  WORD  ScrAspectX;
  WORD  ScrAspectY;
  WORD  PrnAspectX;
  WORD  PrnAspectY;
  WORD  dxPrinter;
  WORD  dyPrinter;
  WORD  AspCorX;
  WORD  AspCorY;
  WORD  wCheck;
  WORD  res1;
  WORD  res2;
  WORD  res3;
  WORD  res4;
} FILEHDR;
```

The first 32 bytes contain the header information described by the above structure:

```
Bytes   Name     Win 1.0   Win 2.0          Win 3.0
```

```
-----   ----     -------   -------           -------

0 - 1   Key1     0x6144    0x694C      |0x6144| or |0x694C|
2 - 3   Key2     0x4D6E    0x536E      |0x4D6E|    |0x536E|

4 - 5   dxFile      X-dimension size of bitmap, in pixels

6 - 7   dyFile      Y-dimension size of bitmap, in pixels

8 - 9   SrcAspectX  Aspect ratio of the screen for which file was
                    created. Obtained from GetDeviceCaps(hScreenDC,
                    ASPECTX);

10 - 11 SrcAspectY  Aspect ratio of the screen for which file was
                    created. Obtained from GetDeviceCaps(hScreenDC,
                    ASPECTY);

12 - 13 PrnAspectX  Aspect ratio of the printer for which file was
                    created. Obtained from GetDeviceCaps(hPrinterDC,
                    ASPECTX);

14 - 15 PrnAspectY  Aspect ratio of the printer for which file was
                    created. Obtained from GetDeviceCaps(hPrinterDC,
                    ASPECTY);

16 - 17 dxPrinter   Resolution of the printer for which file was
                    created. Contains the number of pixels in the x
                    direction on the printer. The selected paper and
                    printer orientation affect this value.

18 - 19 dyPrinter   Resolution of the printer for which file was
                    created. Contains the number of pixels in the y
                    direction on the printer. The selected paper and
                    printer orientation affect this value.

20 - 21 AspCorX     Not used. Set to zero.

22 - 23 AspCorY     Not used. Set to zero.

24 - 25 wCheck      Checksum field. Makes sure that this file is Paint
                    format.

26 - 31 Reserved
```

Index

The Paintbrush bitmap is run-length encoded. Each scan line is encoded separately. The length of the resulting encoded string depends on the pattern of the bitmap. There is an index table following the header. Each entry of this index corresponds to a scan line. It is an unsigned integer and shows the length of the encoded data. Following this table is the encoded data.

```
   32 - xxx     xxx = 32 + sizeof(unsigned int) * dyPrinter
```

ALGORITHM FOR ENCODING

Paint encoding is read in BYTES. There are two types of encoded data: one starts with 0 (zero), the other starts with a nonzero value.

If the data starts with 0, it is a repeated byte pattern. The byte following the 0 tells how many times to repeat the pattern and the subsequent byte is the pattern itself. For example, 0, 0x80, 0xff, should be expanded to eighty (hex) bytes of 0xff.

If the data starts with a nonzero value, X, the following X number of bytes contain unmodified bitmap data, without recognizable pattern. Each scan line can be composed of several of the above mentioned groups of data. Of course, the groups should alternate, meaning that a scan line can start with a repeated pattern followed by some random pattern, which is followed by another repeated pattern.

To reach the data for the nth scan line, add up the values in the index table from entry 0 to entry n - 1.

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrFmt

INF: Group File Format for Windows Program Manager Version 3.0

Article ID: Q66504

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article documents the file format used by version 3.0 of the Microsoft Windows Program Manager to store the icons in each group window. Note that the format of files with the .GRP extension will change in future releases of Windows, including Windows versions 3.1 and later.

More Information:

General Layout

The Program Manager group consists of a header and a variable-size data space.

Program Group Header

The header identifies the group and specifies some global information. Its structure is as follows:

```
     struct tagGROUPHEADER {
          char identifier[4];
          WORD wCheckSum;
          WORD cbGroup;
          WORD nCmdShow;
          RECT rcNormal;
          POINT ptMin;
          WORD pName;
          WORD wLogPixelsX;
          WORD wLogPixelsY;
          WORD wBitsPerPixel;
          WORD wPlanes;
          WORD cItems;
          WORD rgiItems[cItems];
         };
```

The identifier field contains the string "PMCC". If this string is not present in the first 4 bytes of a file, that file will not be a valid group file. The wCheckSum field is used as an additional check on the validity of the file: the sum of the WORDs in the file must be 0. When writing a group file, set wCheckSum accordingly.

The cbGroup field contains the size of the group file. When you are reading a group file, if the file is larger than cbGroup, ignore the extra information. When you are writing a file, set cbGroup to the

size of the file.

The nCmdShow field contains an index used with the ShowWindow() function that specifies whether the group is minimized (SW_SHOWMINIMIZE), normal (SW_SHOWNORMAL), or maximized (SW_SHOWMAXIMIZE).

The rcNormal rectangle specifies the window rectangle when the window is in the normal position (not minimized or maximized). Coordinates are relative to the Program Manager's multiple-document interface (MDI) client window. The point ptMin specifies the position at which the group file icon will be placed when the group is minimized. The pName field is the offset in the file of the zero-terminated title for the group. This title appears in the group's caption.

The wLogPixelsX, wLogPixelsY, wBitsPerPixel, and wPlanes fields contain the corresponding values returned from the GetDeviceCaps() function for a screen (or window) DC at the time the icons in this group file were extracted. If the values do not match the current display when a group is loaded, the Program Manager extracts the appropriate icons from the .EXE files.

The cItems field contains the size of the rgiItems field. Each entry in rgiItems, if nonzero, points to an item structure. Note that cItems is not necessarily the number of items in the group because there may be NULL entries in rgiItems. The first item in rgiItems is the item that is active when the group is initially loaded.

Item Data

Item data may appear anywhere after the group header up to the limit of cbGroup. The offsets in rgiItems point to the following structure:

```
     struct tagITEMINFO {
          POINT pt;
          WORD iIcon;
          WORD cbHeader;
          WORD cbANDPlane;
          WORD cbXORPlane;
          WORD pHeader;
          WORD pANDPlane;
          WORD pXORPlane;
          WORD pName;
          WORD pCommand;
          WORD pIconPath;
         };
```

The pt member specifies the location of the item's icon within the group window. The cbHeader, cbANDPlane, and cbXORPlane members specify the sizes of each of the three icon components. These three components are stored at the offsets specified by the pHeader, pANDPlane, and pXORPlane members. The pHeader member points to a CURSORSHAPE data structure, which is documented below. The name of the item, the command line that it executes, and the path to the file containing its icon are specified in pName, pCommand, and pIconPath, respectively. Each of these members contains the offset from the beginning of the

file to a zero-terminated string.

The iIcon member selects the icon within the application source (.EXE) file if there is more than one icon in the file. It is zero for the first icon in the file. This index has nothing to do with the identifier of the icon resource; it is based upon the order in which icons appear in the file. When adding an icon to a group, append its data to the end of the item-information section and add the appropriate pointers. When an item is deleted, move up any information stored below the deleted icon to shrink the size of the group file.

The pHeader member points to a CURSORSHAPE data structure that contains information about the dimensions and color format of the icon, as follows:

```
typedef struct tagCURSORSHAPE {
     int xHotSpot;
     int yHotSpot;
     int cx;          // width
     int cy;          // height
     int cbWidth;     // Bytes of data per row
                      // accounting for WORD alignment.
     BYTE bPlanes;
     BYTE bBitsPixel;
} CURSORSHAPE;
```

Maintaining Compatibility

An application that reads and writes group files must maintain compatibility with existing and future versions of Program Manager by carefully adhering to a few rules.

The first rule is to use the Program Manager DDE (dynamic data exchange) interface whenever possible. This interface allows an application to add groups and items without dealing with the group file format. Remember to treat direct access to the group files as a measure of last resort. These files were not originally designed to be accessed by programs other than Program Manager.

If the identifier string in the header is not correct, the application should not load the file because it is not a group file. If the checksum is not correct, assume that the file is damaged. If the file is smaller than the header size or the value of cbGroup, the file is damaged. If the file is larger than cbGroup, checksum the additional data if the application performs the checksum; otherwise, ignore the additional data.

Never write a group file that is larger than cbGroup bytes long.

Do not leave "unused" space between structures in the icon data area. Because Program Manager does not recover this space, it will be wasted. This includes space at the end of the file (but less than cbGroup).

Do not attempt to add structures and data for private use because the Program Manager will not maintain the association between private data

and the item if the user edits or deletes the item. The data will be lost and will waste space in the group file.

If the screen metric fields do not match the metrics of the current display driver, do not use any of the icon data. If the application updates the screen metrics, make sure all icons in the group are in the appropriate format. Icons added to the group must match the format specified by the screen metrics fields.

Do not update a group manually while Program Manager is running. To modify a group while Program Manager is running, use the DDE commands documented elsewhere. Updating groups while Program Manager is running can cause the Program Manager to crash.

Additional reference words: 3.00 progman
KBCategory:
KBSubcategory: UsrFmtGrp

INF: Windows Accessories Binary File Formats
Article ID: Q65120

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:
Note: This article is part of a set of seven articles, collectively called the "Windows 3.00 Developer's Notes." More information about the contents of the other articles, and procedures for ordering a hard-copy set, can be found in the knowledge base article titled "The Windows 3.00 Developer's Notes" (Q65260).

This article can be found in the Software/Data Library by searching on the keyword APLETFMT or S12685.

More Information:

This note describes the binary file formats used by Microsoft(R) Windows(TM) Write and Microsoft Windows Calendar.

==

WINDOWS WRITE BINARY FILE FORMAT

==

A Write binary file consists of three parts: The header, the text and picture part, and the formatting information.

HEADER PORTION

The header contains "magic words" and pointers to the subdivisions of the formatting section, as well as information about the length of the file:

Word	Name	Meaning
0	wIdent	Must be 0137061 octal
1	dty	Must be 0
2	wTool	Must be 0125400 octal
3		Reserved; must be 0
4		Reserved; must be 0
5		Reserved; must be 0
6		Reserved; must be 0
7,8	fcMac	Number of bytes of actual text plus 128 (bytes in one sector) [low order word first]
9	pnPara	Page number of start of paragraph information

10	pnFntb	Page number of FNTB (or pnSep if none)
11	pnSep	Page number of SEP (or pnSetb if none)
12	pnSetb	Page number of SETB (or pnPgtb if none)
13	pnPgtb	Page number of PGTB (or pnFfntb if none)
14	pnFfntb	Page number of FFNTB (or pnMac if none)
16..47	szSsht	Not used by Write; reserved for Microsoft Word compatibility
48	pnMac3	Count of pages in whole file (last page number + 1)

The starting page number of character info (pnChar), is not stored, but is computable: pnchar = (fcMac + 127) / 128.

A "page number," as referred to above, means an offset in 128-byte blocks from the start of the file. For example, if pnPara equals 10, the paragraph info is at offset 10 x 128 = 1280 in the file.

In Word files, word 14 of the header is pnMac, and word 48 (pnFfntb in Write files) is zero. Examining the value of word 48 of the header is a good way to tell Write files from Word files.

TEXT AND PICTURE PORTION

The text of the file starts at word 64 (page 1). It is ANSI text (except for pictures), with the following restrictions:

- Paragraph ends are stored as <Return><Linefeed> (ASCII 13, 10). No other occurrences of these two characters are allowed.

- Explicit page breaks are stored as ASCII 12.

- Other line breaks or word wrap information is not stored.

- Tab characters are ASCII 9 (normal).

PICTURES

Pictures are stored as a sequence of bytes in the text stream. These bytes are recognizable as picture information by looking at their paragraph formatting. One picture is exactly one paragraph. Paragraphs that are pictures have a special bit set in their PAP structure (see the description of the paragraph formatting part of the file later on in this document).

Each picture consists of a descriptive header, followed by the bytes of the bitmap or metafile that make up the picture. The picture header is as follows:

Byte	Name	Meaning
0..7	mfp	Windows METAFILEPICT structure (hMF field undefined).

```
   8..9       dxaOffset    Offset of picture from left margin, in
                           twips (1/1440 inch)
   10..11     dxaSize      Horizontal size in twips.
   12..13     dyaSize      Vertical size in twips.
   14..15     cbOldSize    Number of following bytes (actual metafile
                           or bitmap bits). Set to 0.
   16..29     bm           Additional information for bitmaps only.
   30..31     cbHeader     Number of bytes in this header.
   32..35     cbSize       Number of following bytes (actual metafile
                           or bitmap bits). This field replaces
                           cbOldSize for new files.
   36..37     mx           Scaling factor (x).
   38..39     my           Scaling factor (y).
   cbHeader..              Picture contents.
      ..cbHeader+cbSize-1
```

For information about the METAFILEPICT data structure and bitmaps, see WINDOWS.H, the "Microsoft Windows Software Development Kit Guide to Programming," and the "Microsoft Windows Software Development Kit Reference."

If the picture has never been rescaled with the Size Picture command in Write, the scaling factors in each direction will be 1000 (decimal). If the picture has been resized, the scaling factor will be the percentage of the original size that the picture now is, relative to 1000 equaling 100 percent.

The last set of bytes will be bitmap bits if the mm field of the mfp structure (bytes 0..1) is 99 (decimal). This is a special value used within Write only. Otherwise, the bytes will be metafile contents.

FORMATTING PORTION

Character and Paragraph

Both the character and paragraph sections are structured as a set of pages. Each page has the following format:

```
   Byte       Name         Meaning
   ----       ----         -------

   0..3       fcFirst      Byte number in file of first character covered
                           by this page of formatting information. (The
                           byte number of the first character in the text
                           is 128.) [Low order byte first.]
   4..n       rgfod        An array of FODs (see below).
   n+1..126   grpfprop     A group of FPROPs (see below).
   127        cfod         Number of FODs on this page.
```

The structure of each FOD (FOrmat Descriptor) is as follows (these are fixed size):

```
   Word       Name         Meaning
   ----       ----         -------
```

Byte	Name	Meaning
0..1	fcLim	Byte number after last character covered by this FOD.
2	bfprop	Byte offset from beginning of rgfod to corresponding FPROP for these characters or this paragraph.

The structure of each FPROP (Formatting PROPerty) is as follows (these are variable size):

Byte	Name	Meaning
0	cch	Number of bytes in this FPROP.
1..n	rgchProp	A prefix of a CHP (for characters) or a PAP (for paragraphs) sufficient to include all bits that differ from the standard CHP or PAP.

Here is the format of a CHP (CHaracter Property):

Byte	Bit	Name	Meaning
0			Reserved; ignored by Write.
1	0	fBold	Characters are bold.
	1	fItalic	Characters are italic.
	2..7	ftc	Font code (low bits); an index into the FFNTB.
2		hps	Size of font in half pts (standard is 24).
3	0	fUline	Characters are underlined.
	1	fStrike	Reserved; ignored by Write.
	2	fDline	Reserved; ignored by Write.
	3	fOverset	Reserved; ignored by Write.
	4..5	csm	Reserved; ignored by Write.
	6	fSpecial	Set for "(page)" only.
	7		Reserved; ignored by Write.
4	0..2	ftcXtra	Font code (high-order bits, concatenated with ftc).
	3	fOutline	Reserved; ignored by Write.
	4	fShadow	Reserved; ignored by Write.
	5..7		Reserved; ignored by Write.
5		hpsPos	0 for normal; 1..127 for superscript; 128..255 for subscript.

The standard CHP has byte 0 = 1, byte 2 = 24, all other bytes = 0. Note, therefore, that each character FPROP must have a cch of at least 1.

Here is the format of a PAP (PAragraph Property):

Byte	Bit	Name	Meaning
0			Reserved; must be 0.
1	0..1	jc	Justification: 0 = left, 1 = center,

			2 = right, 3 = both.
	2..7		Reserved; must be 0.
2			Reserved; must be 0.
3			Reserved; must be 0.
4..5		dxaRight	Right indent in 20ths of a point.
6..7		dxaLeft	Left indent in 20ths of a point.
8..9		dxaLeft1	First line left indent (relative to dxaLeft).
10..11		dyaLine	Inter-line spacing (standard is 240).
12..13		dyaBefore	Reserved; ignored by Write (standard is 0).
14..15		dyaAfter	Reserved; ignored by Write (standard is 0).
16	0	rhcPage	0 = header, 1 = footer.
	1..2		Reserved; 0 = normal paragraph, nonzero = header or footer paragraph.
	3	rhcFirst	1 = print on first page, 0 = do not print on first.
	4	fGraphics	1 = paragraph is a picture, 0 = paragraph is text.
	5..7		Reserved; must be 0.
17..21			Reserved; must be 0.
22..78			Up to 14 TBDs (tab descriptors; see below).

The format of the TBD (TaB Descriptor) is as follows:

Byte	Bit	Name	Meaning
0..1		dxa	Indent from left margin of tab stop (in 20ths of a point).
2	0..2	jcTab	0 for normal tabs; 3 for decimal tabs
	3..5	tlc	Reserved; ignored by Write.
	6..7		Reserved; must be 0.
3		chAlign	Reserved; ignored by Write.

The standard PAP has byte 0 = 61, byte 2 = 30, bytes 10..11 = (word)240, all other bytes = 0. Note, therefore, that each paragraph FPROP >= 1.

In constructing the pages of formatting information, there is a difference between paragraph and character FODs. A character FOD may describe any number of consecutive characters with the same formatting. However, there must be exactly one paragraph FOD for each text paragraph. In either case, it is allowable, and encouraged, to have multiple FODs point to the same FPROP on a given page. No FOD may point off its page.

There must be no "holes" in either the character or paragraph formatting information; each must begin with the first text character (byte 128) and continue through the last. Therefore, the last character FOD and the last paragraph FOD must have fcLim = fcMac as defined in the header.

Write files have all of their header and footer paragraphs at the

beginning of the document. All header and footer paragraphs appear prior to any normal paragraphs. When reading files created by Word, Write will only recognize those headers and footers that appear at the beginning of the document; all others will be treated as normal text.

Footnotes and Sections

Write documents do not have a footnote table (that is, pnFntb is always equal to pnSep).

A Write document has only one section. If the section properties of a Write document differ from the defaults, the document will contain a SEP section and a SETB section. If not, then neither section is present and pnSep and pnSetb are both equal to pnPgtb.

The format of the SEP (SEction Property) is as follows:

Byte	Name	Meaning
0	cch	Count of bytes used, excluding this byte. All properties at byte positions greater than cch will be set to their standard values.
1..2		Reserved; must be 0.
3..4	yaMac	Page length in 20ths of a point (standard is 11 x 1440 = 15840).
5..6	xaMac	Page width in 20ths of a point (standard is 8.5 x 1440 = 12240).
7..8		Reserved; must be FFFFH.
9..10	yaTop	Top margin in 20ths of a point (standard is 1440).
11..12	dyaText	Height of text in 20ths of a point (standard is 9 x 1440 = 12960).
13..14	xaLeft	Left margin in 20ths of a point (standard is 1.25 x 1440 = 1800).
15..16	dxaText	Width of text area in 20ths of a point (standard is 6 x 1440 = 8640).

As a result:

yaTop + dyaText + (bottom margin, not stored) = yaMac

and

xaLeft + dxaText + (right margin, not stored) = xaMac

Note that if all of the above properties are standard, no SEP or SETB is needed at all. Otherwise, 1 <= cch <= 16.

The structure of the SETB (SEction TaBle) section is as follows:

Word	Name	Meaning
0	csed	Number of sections (always 2 for Write documents, see below).
1	csedMax	Undefined.

2..n	rgsed	Array of SEDs plus zero-padding to fill the sector.

The structure of a SED (SEction Descriptor) is as follows:

Word	Name	Meaning
0..1	cp	Byte address of first character following section.
2	fn	Undefined.
3..4	fcSep	Byte address of associated SEP.

Write documents always have exactly two SED entries. The cp of the first entry indicates that it affects all of the characters in the document. The fcSep of the first entry points to the one SEP in the file. The second SED entry is a dummy, with fcSep set to FFFFFFFFH.

The PGTB section (optional) is on the page immediately after the SEP section.

Note: Contrary to the usage in the preceding portion of this document, the term "page" used in the rest of this section refers to printed pages of a Write document, not 128-byte "pages" of a disk file.

The structure of the PGTB (PaGe TaBle) is as follows:

Word	Name	Meaning
0	cpgd	Number of PGDs (1 or more).
1	cpgdMac	Undefined.
2..n	rgpgd	Array of PGDs plus zero padding to fill the sector.

The structure of a PGD (PaGe Descriptor) is as follows:

Word	Name	Meaning
0	pgn	Page number in printed word documents.
1..2	cpMin	Byte address of first character in printed page.

Font Table

The structure of the FFNTB (Font Face Name TaBle) is as follows:

Byte	Name	Meaning
0..1	cffn	Number of FFNs.
2..n	grpffn	List of FFNs.

The structure of an FFN (Font Face Name) entry in the grpffn is as follows:

Byte	Name	Meaning
0..1	cbFfn	Number of bytes following in this FFN (not including these two bytes).
2	ffid	Font Family ID (see below).
3..cbffn+2	szFfn	Font name (variable length; null-terminated)

A cbFfn value of FFFFH means that the next FFN entry will be found at the start of the next 128-byte page. A cbFfn value of 0 means that there are no more FFN entries in the table.

Possible values for ffid are FF_DONTCARE, FF_ROMAN, FF_SWISS, FF_MODERN, FF_SCRIPT, and FF_DECORATIVE. These are full constants defined in WINDOWS.H. Additional values may be added to the list in future versions of Windows.

==

WINDOWS CALENDAR FILE FORMAT

==

This section describes the file format for the Windows Calendar program supplied with Microsoft Windows version 3.00.

THE HEADER

The first 8 bytes are a character array that identifies the file as a calendar file. This array contains:

```
'C' + 'r' = b5
'A' + 'a' = a2
'L' + 'd' = b0
'E' + 'n' = b3
'N' + 'e' = b3
'D' + 'l' = b0
'A' + 'a' = a2
'R' + 'c' = b5
```

The next 2 bytes contain the integer count of dates described in the file (cDateDescriptors).

The next 12 bytes contain six 2-byte fields of information that is global to the entire file. These variables are normally set by the user through the Alarm Controls and Options Day dialog boxes:

```
WORD MinEarlyRing;            /* early ring in minutes            */
BOOL fSound;                  /* whether alarms should be audible */
int  interval;                /* interval between appointments:
                                       0 = 15 minutes
                                       1 = 30 minutes
                                       2 = 60 minutes             */
```

```
int  mininterval;              /* the interval in minutes          */
BOOL f24HourFormat;            /* TRUE for 24 hour format,
                                  0 otherwise                      */
int   StartTime;               /* starting time in Day Mode, i.e.,
                                  the time that normally appears
                                  first in the display, in
                                  minutes past midnight */
```

The rest of the first 64 bytes are reserved.

THE DATE DESCRIPTORS
====================

The date descriptor array appears next. Each entry in the array describes one day. cDateDescriptors (described above) is the number of entries in the array. Each element in the array consists of 12 bytes, in six 2-byte fields, as follows:

```
unsigned Date;                 /* the date, in days past 1/1/1980  */
int      fMarked;              /* bitmask indicating which mark(s)
                                  are set for date:
                                      box         =  128
                                      parentheses =  256
                                      circle      =  512
                                      cross       = 1024
                                      underscore  = 2048           */
int      cAlarms;              /* number of alarms set for the day */
unsigned FileBlockOffset;      /* file offset, in 64 byte blocks,
                                  of where the day's information is
                                  stored. Only the low 15 bits are
                                  used (the high bit will be 0).
                                  Thus, if this offset were equal
                                  to 6, the day's information would
                                  be stored at byte 6*64 in the
                                  file. */
int      reserved;             /* this equals 0xfff                */
unsigned reserved;             /* this equals 0xfff                */
```

DAY-SPECIFIC INFORMATION
========================

All day information is stored after the date descriptor array, on even 64 byte boundaries. Day information is stored as follows:

```
unsigned reserved;             /* this must equal 0                */
unsigned Date;                 /* in days past 1/1/1980            */
unsigned reserved;             /* this must equal 1                */
unsigned cbNotes;              /* the number of bytes of note
                                  information, including the null.
                                  This information appears in the
                                  note array below the appointment
                                  list. */
unsigned cbAppointment;        /* count of bytes of appointment
                                  information                      */
char     Notes [cbNotes];      /* the text info of the note        */
```

```
BYTE     ApptInfo [];         /* the block of appointments        */
```

APPOINTMENT-SPECIFIC INFORMATION
================================

The information in the appointment block is stored as a list of single appointments. Each appointment consists of the following fields:

```
char    cBytes;               /* the count of bytes in the
                                 appointment, e.g. the next
                                 appointment's info will be at
                                 & (ThisAppt.cBytes) +
                                 ThisAppt.cBytes                  */
char    flags;                /* the low bit controls the alarm
                                 (1 = alarm), while bit 1, if set,
                                 indicates that it is a special
                                 time                             */
int     Time;                 /* in minutes past midnight */
char    ApptDesc[];           /* a null-terminated string which
                                 contains the text associated with
                                 an appointment */
```

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrFmtGeneral

INF: Executable-File Header Format
Article ID: Q65122

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:
Note: This article is part of a set of seven articles, collectively called the "Windows 3.00 Developer's Notes." More information about the contents of the other articles, and procedures for ordering a hard-copy set, can be found in the knowledge base article titled "INF: The Windows 3.00 Developer's Notes" (Q65260).

This article can be found in the Software/Data Library by searching on the word EXEFMT or S12688. EXEFMT was archived using the PKware file-compression utility.

More Information:

Microsoft defined the segmented executable file format for Windows applications and dynamic-link libraries (DLLs). This file format is also referred to as the New Executable Format. This new format is an extension of the existing MS-DOS .EXE format (old-style format). The purpose of the segmented executable format is to provide the information needed to support the dynamic linking and segmentation capabilities of the Windows environment.

An executable file contains Microsoft Windows code and data, or Windows code, data, and resources. Specific fields have been added to the old-style .EXE format header to indicate the existence of the segmented file format. The old-style header may contain a valid executable program, called a stub program, that will be executed if the program is run on MS-DOS (without Windows). This stub program usually prints a message indicating that Microsoft Windows is required to run the program. The segmented executable format extensions also begin with a header that describes the contents and location of the executable image in the file. The loader uses this header information when it loads the executable segments in memory.

===
OLD-STYLE HEADER EXTENSIONS
===

The old-style header contains information the loader expects for a DOS executable file. It describes a stub program (WINSTUB) the loader can place in memory when necessary, it points to the new-style header, and it contains the stub programs relocation table.

The following illustrates the distinct parts of the old-style executable format:

```
     +-------------------------+
 00h |  Old-style header info  |
     +-------------------------+
 20h |        Reserved         |
     +-------------------------+
 3Ch |   Offset to segmented   |
     |       .EXE header       |
     +-------------------------+
 40h |  Relocation table and   |
     |    DOS stub program     |
     +-------------------------+
     |  Segmented .EXE Header  |
     |            .            |
     |            .            |
     |            .            |
```

The word at offset 18h in the old-style .EXE header contains the relative byte offset to the stub program's relocation table. If this offset is 40h, then the double word at offset 3Ch is assumed to be the relative byte offset from the beginning of the file to the beginning of the segmented executable header. A new-format .EXE file is identified if the segmented executable header contains a valid signature. If the signature is not valid, the file is assumed to be an old-style format .EXE file. The remainder of the old-style format header will describe a DOS program, the stub. The stub may be any valid program but will typically be a program that displays an error message.

==

SEGMENTED EXE FORMAT

==

Because Windows executable files are often larger than one segment (64K), additional information (that does not appear in the old-style header) is required so that the loader can load each segment properly. The segmented EXE format was developed to provide the loader with this information.

The segmented .EXE file has the following format:

```
     +------------------+
 00h |  Old-style EXE   |
     |      Header      |
     +------------------+
 20h |    Reserved      |
     +------------------+
 3Ch |    Offset to     | ---+
     | Segmented Header|    |
     +------------------+    |
 40h | Relocation Table|    |
     |  & Stub Program |    |
     +------------------+    |
     |                  |    |
     +------------------+    |
 xxh |  Segmented EXE   | <--+
     |      Header      |
```

```
+-----------------+
|  Segment Table  |
+-----------------+
| Resource Table  |
+-----------------+
|  Resident Name  |
|      Table      |
+-----------------+
| Module Reference|
|      Table      |
+-----------------+
| Imported Names  |
|      Table      |
+-----------------+
|   Entry Table   |
+-----------------+
|  Non-Resident   |
|   Name Table    |
+-----------------+
|   Seg #1 Data   |
|   Seg #1 Info   |
+-----------------+
        .
        .
        .
+-----------------+
|   Seg #n Data   |
|   Seg #n Info   |
+-----------------+
```

The following sections describe each of the components that make up the segmented EXE format. Each section contains a description of the component and the fields in the structures that make up that component.

Note: All unused fields and flag bits are reserved for future use and must contain 0 (zero) values.

SEGMENTED EXE HEADER

The segmented EXE header contains general information about the EXE file and contains information on the location and size of the other sections. The Windows loader copies this section, along with other data, into the module table in the system data. The module table is internal data used by the loader to manage the loaded executable modules in the system and to support dynamic linking.

The following describes the format of the segmented executable header. For each field, the offset is given relative to the beginning of the segmented header, the size of the field is defined, and a description is given.

```
Offset Size Description
------ ---- -----------
```

```
00h     DW  Signature word.
            "N" is low-order byte.
            "E" is high-order byte.

02h     DB  Version number of the linker.

03h     DB  Revision number of the linker.

04h     DW  Entry Table file offset, relative to the beginning of
            the segmented EXE header.
06h     DW  Number of bytes in the entry table.

08h     DD  32-bit CRC of entire contents of file.
            These words are taken as 00 during the calculation.

0Ch     DW  Flag word.
            0000h = NOAUTODATA
            0001h = SINGLEDATA (Shared automatic data segment)
            0002h = MULTIPLEDATA (Instanced automatic data
                    segment)
            2000h = Errors detected at link time, module will not
                    load.
            8000h = Library module.
                    The SS:SP information is invalid, CS:IP points
                    to an initialization procedure that is called
                    with AX equal to the module handle. This
                    initialization procedure must perform a far
                    return to the caller, with AX not equal to
                    zero to indicate success, or AX equal to zero
                    to indicate failure to initialize. DS is set
                    to the library's data segment if the
                    SINGLEDATA flag is set. Otherwise, DS is set
                    to the caller's data segment.

                    A program or DLL can only contain dynamic
                    links to executable files that have this
                    library module flag set. One program cannot
                    dynamic-link to another program.

0Eh     DW  Segment number of automatic data segment.
            This value is set to zero if SINGLEDATA and
            MULTIPLEDATA flag bits are clear, NOAUTODATA is
            indicated in the flags word.

            A Segment number is an index into the module's segment
            table. The first entry in the segment table is segment
            number 1.

10h     DW  Initial size, in bytes, of dynamic heap added to the
            data segment. This value is zero if no initial local
            heap is allocated.

12h     DW  Initial size, in bytes, of stack added to the data
            segment. This value is zero to indicate no initial
            stack allocation, or when SS is not equal to DS.
```

```
    14h     DD  Segment number:offset of CS:IP.

    18h     DD  Segment number:offset of SS:SP.
                If SS equals the automatic data segment and SP equals
                zero, the stack pointer is set to the top of the
                automatic data segment just below the additional heap
                area.

                    +--------------------------+
                    | additional dynamic heap  |
                    +--------------------------+ <- SP
                    |    additional stack      |
                    +--------------------------+
                    | loaded auto data segment |
                    +--------------------------+ <- DS, SS

    1Ch     DW  Number of entries in the Segment Table.

    1Eh     DW  Number of entries in the Module Reference Table.
    20h     DW  Number of bytes in the Non-Resident Name Table.

    22h     DW  Segment Table file offset, relative to the beginning
                of the segmented EXE header.

    24h     DW  Resource Table file offset, relative to the beginning
                of the segmented EXE header.

    26h     DW  Resident Name Table file offset, relative to the
                beginning of the segmented EXE header.

    28h     DW  Module Reference Table file offset, relative to the
                beginning of the segmented EXE header.

    2Ah     DW  Imported Names Table file offset, relative to the
                beginning of the segmented EXE header.

    2Ch     DD  Non-Resident Name Table offset, relative to the
                beginning of the file.

    30h     DW  Number of movable entries in the Entry Table.

    32h     DW  Logical sector alignment shift count, log(base 2) of
                the segment sector size (default 9).

    34h     DW  Number of resource entries.

    36h     DB  Executable type, used by loader.
                  02h = WINDOWS

    37h-3Fh DB  Reserved, currently 0's.
```

==

SEGMENT TABLE

==

The segment table contains an entry for each segment in the executable

file. The number of segment table entries are defined in the segmented EXE header. The first entry in the segment table is segment number 1. The following is the structure of a segment table entry.

Size	Description
DW	Logical-sector offset (n byte) to the contents of the segment data, relative to the beginning of the file. Zero means no file data.
DW	Length of the segment in the file, in bytes. Zero means 64K.
DW	Flag word.
DW	Minimum allocation size of the segment, in bytes. Total size of the segment. Zero means 64K.

Flag word values:

Value	Name	Meaning
0007h	TYPE_MASK	Segment-type field.
0000h	CODE	Code-segment type.
0001h	DATA	Data-segment type.
0010h	MOVEABLE	Segment is not fixed.
0040h	PRELOAD	Segment will be preloaded; read-only if this is a data segment.
0100h	RELOCINFO	Set if segment has relocation records.
F000h	DISCARD	Discard priority.

RESOURCE TABLE

The resource table follows the segment table and contains entries for each resource in the executable file. The resource table consists of an alignment shift count, followed by a table of resource records. The resource records define the type ID for a set of resources. Each resource record contains a table of resource entries of the defined type. The resource entry defines the resource ID or name ID for the resource. It also defines the location and size of the resource. The following describes the contents of each of these structures:

Size	Description
DW	Alignment shift count for resource data.

A table of resource type information blocks follows. The following is the format of each type information block:

Size	Description
DW	Type ID. This is an integer type if the high-order bit is set (8000h); otherwise, it is an offset to the type string, the offset is relative to the beginning of the resource table. A zero type ID marks the end of the resource type information blocks.
DW	Number of resources for this type.
DD	Reserved.

```
        A table of resources for this type follows. The following is
        the format of each resource (8 bytes each):

            DW  File offset to the contents of the resource data,
                relative to beginning of file. The offset is in terms
                of the alignment shift count value specified at
                beginning of the resource table.

            DW  Length of the resource in the file (in bytes).

            DW  Flag word.
                0010h = MOVEABLE  Resource is not fixed.
                0020h = PURE      Resource can be shared.
                0040h = PRELOAD   Resource is preloaded.

            DW  Resource ID. This is an integer type if the high-order
                bit is set (8000h), otherwise it is the offset to the
                resource string, the offset is relative to the
                beginning of the resource table.

            DD  Reserved.

   Resource type and name strings are stored at the end of the
   resource table. Note that these strings are NOT null terminated and
   are case sensitive.

   DB   Length of the type or name string that follows. A zero value
        indicates the end of the resource type and name string, also
        the end of the resource table.

   DB   ASCII text of the type or name string.
```

==

RESIDENT-NAME TABLE

==

The resident-name table follows the resource table, and contains this module's name string and resident exported procedure name strings. The first string in this table is this module's name. These name strings are case-sensitive and are not null-terminated. The following describes the format of the name strings:

```
   Size Description
   ---- -----------

   DB   Length of the name string that follows. A zero value indicates
        the end of the name table.

   DB   ASCII text of the name string.

   DW   Ordinal number (index into entry table). This value is ignored
        for the module name.
```

==

MODULE-REFERENCE TABLE

The module-reference table follows the resident-name table. Each entry contains an offset for the module-name string within the imported-names table; each entry is 2 bytes long.

Size	Description
DW	Offset within Imported Names Table to referenced module name string.

IMPORTED-NAME TABLE

The imported-name table follows the module-reference table. This table contains the names of modules and procedures that are imported by the executable file. Each entry is composed of a 1-byte field that contains the length of the string, followed by any number of characters. The strings are not null-terminated and are case sensitive.

Size	Description
DB	Length of the name string that follows.
DB	ASCII text of the name string.

ENTRY TABLE

The entry table follows the imported-name table. This table contains bundles of entry-point definitions. Bundling is done to save space in the entry table. The entry table is accessed by an ordinal value. Ordinal number one is defined to index the first entry in the entry table. To find an entry point, the bundles are scanned searching for a specific entry point using an ordinal number. The ordinal number is adjusted as each bundle is checked. When the bundle that contains the entry point is found, the ordinal number is multiplied by the size of the bundle's entries to index the proper entry.

The linker forms bundles in the most dense manner it can, under the restriction that it cannot reorder entry points to improve bundling. The reason for this restriction is that other .EXE files may refer to entry points within this bundle by their ordinal number. The following describes the format of the entry table bundles.

Size	Description
DB	Number of entries in this bundle. All records in one bundle

```
        are either moveable or refer to the same fixed segment. A zero
        value in this field indicates the end of the entry table.

   DB   Segment indicator for this bundle. This defines the type of
        entry table entry data within the bundle. There are three
        types of entries that are defined.

        000h = Unused entries. There is no entry data in an unused
               bundle. The next bundle follows this field. This is
               used by the linker to skip ordinal numbers.

        001h-0FEh = Segment number for fixed segment entries. A fixed
               segment entry is 3 bytes long and has the following
               format.

            DB  Flag word.
                01h = Set if the entry is exported.
                02h = Set if the entry uses a global (shared) data
                      segments.
                      The first assembly-language instruction in the
                      entry point prologue must be "MOV AX,data
                      segment number". This may be set only for
                      SINGLEDATA library modules.

            DW  Offset within segment to entry point.

        0FFH = Moveable segment entries. The entry data contains the
               segment number for the entry points. A moveable segment
               entry is 6 bytes long and has the following format.

            DB  Flag word.
                01h = Set if the entry is exported.
                02h = Set if the entry uses a global (shared) data
                      segments.

            INT 3FH.

            DB  Segment number.

            DW  Offset within segment to entry point.
```

==

NONRESIDENT-NAME TABLE

==

The nonresident-name table follows the entry table, and contains a module description and nonresident exported procedure name strings. The first string in this table is a module description. These name strings are case-sensitive and are not null-terminated. The name strings follow the same format as those defined in the resident name table.

==

PER SEGMENT DATA

==

The location and size of the per-segment data is defined in the segment table entry for the segment. If the segment has relocation fixups, as defined in the segment table entry flags, they directly follow the segment data in the file. The relocation fixup information is defined as follows:

```
Size Description
---- -----------

DW   Number of relocation records that follow.

A table of relocation records follows. The following is the format
of each relocation record.

     DB  Source type.
         0Fh = SOURCE_MASK
         00h = LOBYTE
         02h = SEGMENT
         03h = FAR_ADDR (32-bit pointer)
         05h = OFFSET (16-bit offset)

     DB  Flags byte.
         03h = TARGET_MASK
         00h = INTERNALREF
         01h = IMPORTORDINAL
         02h = IMPORTNAME
         03h = OSFIXUP
         04h = ADDITIVE

     DW  Offset within this segment of the source chain.
         If the ADDITIVE flag is set, then target value is added to
         the source contents, instead of replacing the source and
         following the chain. The source chain is an 0FFFFh
         terminated linked list within this segment of all
         references to the target.

     The target value has four types that are defined in the flag
     byte field. The following are the formats for each target
     type:

     INTERNALREF

         DB  Segment number for a fixed segment, or 0FFh for a
             movable segment.

         DB  0

         DW  Offset into segment if fixed segment, or ordinal
             number index into Entry Table if movable segment.

     IMPORTNAME

         DW  Index into module reference table for the imported
             module.
```

```
    DW  Offset within Imported Names Table to procedure name
        string.

IMPORTORDINAL

    DW  Index into module reference table for the imported
        module.
    DW  Procedure ordinal number.

OSFIXUP

    DW  Operating system fixup type.
        Floating-point fixups.
        0001h = FIARQQ, FJARQQ
        0002h = FISRQQ, FJSRQQ
        0003h = FICRQQ, FJCRQQ
        0004h = FIERQQ
        0005h = FIDRQQ
        0006h = FIWRQQ

    DW  0
```

==

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrFmtExe

INF: Font-File Format
Article ID: Q65123

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:
Note: This article is part of a set of seven articles, collectively called the "Windows 3.00 Developer's Notes." More information about the contents of the other articles, and procedures for ordering a hard-copy set, can be found in the knowledge base article titled "INF: The Windows 3.00 Developer's Notes" (Q65260).

This article can be found in the Software/Data Library by searching on the keyword FONTFMT or S12687.

More Information:

Formats for Microsoft Windows font files are defined for both raster and vector fonts. These formats can be used by smart text generators in some GDI support modules. The vector formats, in particular, are more frequently used by GDI itself than by support modules.

Both raster and vector font files begin with information that is common to both, and then continue with information that differs for each type of file.

For Windows 3.00, the font-file header includes six new fields: dFlags, dfAspace, dfBspace, dfCspace, dfColorPointer, and dfReserved1. These fields are not used in Windows 3.00. To ensure compatibility with future versions of Windows, these fields should be set to zero.

All device drivers support the Windows 2.x fonts. However, not all device drivers support the Windows 3.00 version.

Windows 3.00 font files include the glyph table in dfCharTable, which consists of structures that describe the bits for characters in the font file. This version enables fonts to exceed 64K in size, the size limit of Windows 2.x fonts. This is made possible by the use of 32-bit offsets to the character glyphs in dfCharTable.

Because of the 32-bit offsets and their potentially large size, these fonts are designed for use on systems that are running Windows version 3.00 in protected (standard or 386 enhanced) mode with an 80386 (or higher) processor where the processor's 32-bit registers can access the character glyphs. Typically, device drivers use the Windows 3.00 version of a font only when both of these conditions are true.

Font files are stored with an .FNT extension of the form NAME.FNT. The information at the beginning of both raster and vector versions of Windows 3.00 font files is shown in the following list:

```
Field          Description
-----          -----------

dfVersion      2 bytes specifying the version (0200H or 0300H) of
               the file.

dfSize         4 bytes specifying the total size of the file in
               bytes.

dfCopyright    60 bytes specifying copyright information.

dfType         2 bytes specifying the type of font file.

               The low-order byte is exclusively for GDI use. If the
               low-order bit of the WORD is zero, it is a bitmap
               (raster) font file. If the low-order bit is 1, it is a
               vector font file. The second bit is reserved and must
               be zero. If no bits follow in the file and the bits are
               located in memory at a fixed address specified in
               dfBitsOffset, the third bit is set to 1; otherwise, the
               bit is set to 0 (zero). The high-order bit of the low
               byte is set if the font was realized by a device. The
               remaining bits in the low byte are reserved and set to
               zero.

               The high byte is reserved for device use and will
               always be set to zero for GDI-realized standard fonts.
               Physical fonts with the high-order bit of the low byte
               set may use this byte to describe themselves. GDI will
               never inspect the high byte.

dfPoints       2 bytes specifying the nominal point size at which
               this character set looks best.

dfVertRes      2 bytes specifying the nominal vertical resolution
               (dots-per-inch) at which this character set was
               digitized.

dfHorizRes     2 bytes specifying the nominal horizontal resolution
               (dots-per-inch) at which this character set was
               digitized.

dfAscent       2 bytes specifying the distance from the top of a
               character definition cell to the baseline of the
               typographical font. It is useful for aligning the
               baselines of fonts of different heights.

dfInternalLeading
               Specifies the amount of leading inside the bounds set
               by dfPixHeight. Accent marks may occur in this area.
               This may be zero at the designer's option.

dfExternalLeading
               Specifies the amount of extra leading that the designer
               requests the application add between rows. Since this
               area is outside of the font proper, it contains no
```

marks and will not be altered by text output calls in either the OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.

dfItalic — 1 (one) byte specifying whether or not the character definition data represent an italic font. The low-order bit is 1 if the flag is set. All the other bits are zero.

dfUnderline — 1 byte specifying whether or not the character definition data represent an underlined font. The low-order bit is 1 if the flag is set. All the other bits are 0 (zero).

dfStrikeOut — 1 byte specifying whether or not the character definition data represent a struckout font. The low-order bit is 1 if the flag is set. All the other bits are zero.

dfWeight — 2 bytes specifying the weight of the characters in the character definition data, on a scale of 1 to 1000. A dfWeight of 400 specifies a regular weight.

dfCharSet — 1 byte specifying the character set defined by this font.

dfPixWidth — 2 bytes. For vector fonts, specifies the width of the grid on which the font was digitized. For raster fonts, if dfPixWidth is nonzero, it represents the width for all the characters in the bitmap; if it is zero, the font has variable width characters whose widths are specified in the dfCharTable array.

dfPixHeight — 2 bytes specifying the height of the character bitmap (raster fonts), or the height of the grid on which a vector font was digitized.

dfPitchAndFamily

Specifies the pitch and font family. The low bit is set if the font is variable pitch. The high four bits give the family name of the font. Font families describe in a general way the look of a font. They are intended for specifying fonts when the exact face name desired is not available. The families are as follows:

Family	Description
FF_DONTCARE (0<<4)	Don't care or don't know.
FF_ROMAN (1<<4)	Proportionally spaced fonts with serifs.
FF_SWISS (2<<4)	Proportionally spaced fonts without serifs.
FF_MODERN (3<<4)	Fixed-pitch fonts.
FF_SCRIPT (4<<4)	
FF_DECORATIVE (5<<4)	

dfAvgWidth 2 bytes specifying the width of characters in the font. For fixed-pitch fonts, this is the same as dfPixWidth. For variable-pitch fonts, this is the width of the character "X."

dfMaxWidth 2 bytes specifying the maximum pixel width of any character in the font. For fixed-pitch fonts, this is simply dfPixWidth.

dfFirstChar 1 byte specifying the first character code defined by this font. Character definitions are stored only for the characters actually present in a font. Therefore, use this field when calculating indexes into either dfBits or dfCharOffset.

dfLastChar 1 byte specifying the last character code defined by this font. Note that all the characters with codes between dfFirstChar and dfLastChar must be present in the font character definitions.

dfDefaultChar 1 byte specifying the character to substitute whenever a string contains a character out of the range. The character is given relative to dfFirstChar so that dfDefaultChar is the actual value of the character, less dfFirstChar. The dfDefaultChar should indicate a special character that is not a space.

dfBreakChar 1 byte specifying the character that will define word breaks. This character defines word breaks for word wrapping and word spacing justification. The character is given relative to dfFirstChar so that dfBreakChar is the actual value of the character, less that of dfFirstChar. The dfBreakChar is normally (32 - dfFirstChar), which is an ASCII space.

dfWidthBytes 2 bytes specifying the number of bytes in each row of the bitmap. This is always even, so that the rows start on WORD boundaries. For vector fonts, this field has no meaning.

dfDevice 4 bytes specifying the offset in the file to the string giving the device name. For a generic font, this value is zero.

dfFace 4 bytes specifying the offset in the file to the null-terminated string that names the face.

dfBitsPointer 4 bytes specifying the absolute machine address of the bitmap. This is set by GDI at load time. The dfBitsPointer is guaranteed to be even.

dfBitsOffset 4 bytes specifying the offset in the file to the beginning of the bitmap information. If the 04H bit in the dfType is set, then dfBitsOffset is an absolute address of the bitmap (probably in ROM).

For raster fonts, dfBitsOffset points to a sequence of

bytes that make up the bitmap of the font, whose height is the height of the font, and whose width is the sum of the widths of the characters in the font rounded up to the next WORD boundary.

For vector fonts, it points to a string of bytes or words (depending on the size of the grid on which the font was digitized) that specify the strokes for each character of the font. The dfBitsOffset field must be even.

dfReserved — 1 byte, not used.

dfFlags — 4 bytes specifying the bits flags, which are additional flags that define the format of the Glyph bitmap, as follows:

```
DFF_FIXED            equ  0001h ; font is fixed pitch
DFF_PROPORTIONAL     equ  0002h ; font is proportional
                                ; pitch
DFF_ABCFIXED         equ  0004h ; font is an ABC fixed
                                ; font
DFF_ABCPROPORTIONAL  equ  0008h ; font is an ABC pro-
                                ; portional font
DFF_1COLOR           equ  0010h ; font is one color
DFF_16COLOR          equ  0020h ; font is 16 color
DFF_256COLOR         equ  0040h ; font is 256 color
DFF_RGBCOLOR         equ  0080h ; font is RGB color
```

dfAspace — 2 bytes specifying the global A space, if any. The dfAspace is the distance from the current position to the left edge of the bitmap.

dfBspace — 2 bytes specifying the global B space, if any. The dfBspace is the width of the character.

dfCspace — 2 bytes specifying the global C space, if any. The dfCspace is the distance from the right edge of the bitmap to the new current position. The increment of a character is the sum of the three spaces. These apply to all glyphs and is the case for DFF_ABCFIXED.

dfColorPointer — 4 bytes specifying the offset to the color table for color fonts, if any. The format of the bits is similar to a DIB, but without the header. That is, the characters are not split up into disjoint bytes. Instead, they are left intact. If no color table is needed, this entry is NULL.
[NOTE: This information is different from that in the hard-copy Developer's Notes and reflects a correction.]

dfReserved1 — 16 bytes, not used.
[NOTE: This information is different from that in the hard-copy Developer's Notes and reflects a correction.]

dfCharTable — For raster fonts, the CharTable is an array of entries

each consisting of two 2-byte WORDs for Windows 2.x and three 2-byte WORDs for Windows 3.00. The first WORD of each entry is the character width. The second WORD of each entry is the byte offset from the beginning of the FONTINFO structure to the character bitmap. For Windows 3.00, the second and third WORDs are used for the offset.

There is one extra entry at the end of this table that describes an absolute-space character. This entry corresponds to a character that is guaranteed to be blank; this character is not part of the normal character set.

The number of entries in the table is calculated as ((dfLastChar - dfFirstChar) + 2). This includes a spare, the sentinel offset mentioned in the following paragraph.

For fixed-pitch vector fonts, each 2-byte entry in this array specifies the offset from the start of the bitmap to the beginning of the string of stroke specification units for the character. The number of bytes or WORDs to be used for a particular character is calculated by subtracting its entry from the next one, so that there is a sentinel at the end of the array of values.

For proportionally spaced vector fonts, each 4-byte entry is divided into two 2-byte fields. The first field gives the starting offset from the start of the bitmap of the character strokes. The second field gives the pixel width of the character.

<facename> An ASCII character string specifying the name of the font face. The size of this field is the length of the string plus a NULL terminator.

<devicename> An ASCII character string specifying the name of the device if this font file is for a specific device. The size of this field is the length of the string plus a NULL terminator.

<bitmaps> This field contains the character bitmap definitions. Each character is stored as a contiguous set of bytes. (In the old font format, this was not the case.)

The first byte contains the first 8 bits of the first scanline (that is, the top line of the character). The second byte contains the first 8 bits of the second scanline. This continues until a first "column" is completely defined.

The following byte contains the next 8 bits of the first scanline, padded with zeros on the right if necessary (and so on, down through the second "column"). If the glyph is quite narrow, each scanline is covered by 1 byte, with bits set to zero as

necessary for padding. If the glyph is very wide, a third or even fourth set of bytes can be present.

Note: The character bitmaps must be stored contiguously and arranged in ascending order.

The following is a single-character example, in which are given the bytes for a 12 x 14 pixel character, as shown here schematically.

```
............
.....**.....
....*..*....
...*....*...
..*......*..
..*......*..
..*......*..
..********..
..*......*..
..*......*..
..*......*..
............
............
............
```

The bytes are given here in two sets, because the character is less than 17 pixels wide.

```
00 06 09 10 20 20 20 3F 20 20 20 00 00 00
00 00 00 80 40 40 40 C0 40 40 40 00 00 00
```

Note that in the second set of bytes, the second digit of each is always zero. It would correspond to the 13th through 16th pixels on the right side of the character, if they were present.

The Windows 2.x version of dfCharTable has a GlyphEntry structure with the following format:

```
GlyphEntry    struc
geWidth       dw     ?    ; width of character bitmap in pixels
geOffset      dw     ?    ; pointer to the bits
GlyphEntry    ends
```

The Windows 3.00 version of the dfCharTable is dependent on the format of the Glyph bitmap.

Note: The only formats supported in Windows 3.00 will be DFF_FIXED and DFF_PROPORTIONAL.

DFF_FIXED
DFF_PROPORTIONAL

```
GlyphEntry    struc
geWidth       dw     ?    ; width of character bitmap in pixels
geOffset      dd     ?    ; pointer to the bits
```

```
GlyphEntry    ends

DFF_ABCFIXED
DFF_ABCPROPORTIONAL

GlyphEntry    struc
geWidth       dw     ?   ; width of character bitmap in pixels
geOffset      dd     ?   ; pointer to the bits
geAspace      dd     ?   ; A space in fractional pixels (16.16)
geBspace      dd     ?   ; B space in fractional pixels (16.16)
geCspace      dw     ?   ; C space in fractional pixels (16.16)
GlyphEntry    ends
```

The fractional pixels are expressed as a 32-bit signed number with an implicit binary point between bits 15 and 16. This is referred to as a 16.16 ("sixteen dot sixteen") fixed-point number.

The ABC spacing here is the same as that defined above. However, here there are specific sets for each character.

```
DFF_1COLOR
DFF_16COLOR
DFF_256COLOR
DFF_RGBCOLOR

GlyphEntry    struc
geWidth       dw     ?   ; width of character bitmap in pixels
geOffset      dd     ?   ; pointer to the bits
geHeight      dw     ?   ; height of character bitmap in pixels
geAspace      dd     ?   ; A space in fractional pixels (16.16)
geBspace      dd     ?   ; B space in fractional pixels (16.16)
geCspace      dd     ?   ; C space in fractional pixels (16.16)
GlyphEntry    ends
```

DFF_1COLOR means 8 pixels per byte
DFF_16COLOR means 2 pixels per byte
DFF_256COLOR means 1 pixel per byte
DFF_RGBCOLOR means RGBquads

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrFmtGeneral

INF: Corrections to Program Manager Group File Format Docs

Article ID: Q86334

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The text below provides additions and corrections to the documentation of the Microsoft Windows Program Manager group (.GRP) file format. This format is documented in Chapter 5 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 4: Resources" manual and in the associated online help files (WIN31WH.HLP and WIN31QH.HLP) provided with the Microsoft Windows Software Development Kit (SDK) version 3.1.

More Information:

- In the "Organization of a Group File" section, page 61, the third paragraph is incorrect; it says:

  The item data entries are followed by entries that contain the color data for the application icons.

  Group files do not contain color data. The application icons use system colors only, and do not require any color data to be saved in the group file.

- The GROUPHEADER structure is incorrectly documented on page 62. The wBitsPerPixel and wPlanes fields are only one byte long. The correct GROUPHEADER structure is as follows:

```
struct tagGROUPHEADER {
    char  cIdentifier[4];
    WORD  wCheckSum;
    WORD  cbGroup;
    WORD  nCmdShow;
    RECT  rcNormal;
    POINT ptMin;
    WORD  pName;
    WORD  wLogPixelsX;
    WORD  wLogPixelsY;
    BYTE  bBitsPerPixel;
    BYTE  bPlanes;
    WORD  wReserved;  // Should be 0x0000
    WORD  cItems;
    WORD  rgiItems[cItems];
}
```

- The cbGroup field in the GROUPHEADER structure (described on page 62) specifies the size of the group file not including the tag data. This is in accord with Windows 3.0 group files, which do

not contain any tag data. In Windows 3.1 group files, cbGroup can be used as an offset to the tag data.

- The description of the rgiItems field in the GROUPHEADER structure on page 63 is missing two words. The description should be:

  Specifies an array of offsets to ITEMDATA structures.

- Page 64 says that the pHeader field in the ITEMDATA structure points to the resource header for the icon, but fails to specify the structure of the resource header. Its structure is as follows:

```
struct {
    int xHotSpot;    // Should be 0
    int yHotSpot;    // Should be 0
    int cx;          // Icon width
    int cy;          // Icon height
    int cbWidth;     // Bytes per row accounting
                     // for WORD alignment
    BYTE bPlanes;    // Count of planes
    BYTE bBitsPixel; // Bits per pixel
  }
```

- The last TAGDATA structure (described on page 64-65) in the group file will have 0xFFFF as the value for its wID. This simply signals the end of file.

Additional reference words: 3.10 on-line
KBCategory:
KBSubcategory: UsrFmtGrp

INF: Sample Code to Access New EXE File Headers
Article ID: Q42061

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

DUMPDESC is a file in the Software/Data Library that demonstrates reading the description line from an EXE file. During the processing of the WM_PAINT message, the program reads its own description line and displays the contents in its client area.

The most relevant portion of the code is in the DumpDescPaint() function.

DUMPDESC can be found in the Software/Data Library by searching on the word DUMPDESC, the Q number of this article, or S12229. DUMPDESC was archived using the PKware file-compression utility.

Additional reference words: SR# G890224-12701 3.00
KBCategory:
KBSubcategory: UsrFmtExe

INF: Windows 3.1 Card File Format

Article ID: Q99340

Summary:

This article documents the file format used by Microsoft Windows version 3.1 Cardfile. Please note that the Cardfile file format (.CRD) may change in future versions. All numbers in this document, including those in the descriptive text, should be interpreted as hexadecimal numerals. All data pointers and count bytes in the file are unsigned binary/hexadecimal integers in least-to-most significant format. All text in the file is saved in low ASCII format. In the text section of a card's data, <CR> is always followed by <LF>.

More Information:

The Cardfile file format is as follows:

Byte #	Description
0 - 2	Signature bytes--always "RRG" (52 52 47).
3 - 6	Last object's ID.
7 - 8	Number of cards in file.

Beyond the first 9 bytes are the index lines--the information about the top line of each card. The first index entry begins at byte 9 in the file, and successive entries begin 34 bytes after the beginning of the last index entry (the second entry at byte 3D, the third entry at byte 71, and so forth). The format for each index line entry is as follows:

Byte #	Description
0 - 5	Null bytes, reserved for future use (should all be 00).
6 - 9	Absolute position of card data in file.
A	Flag byte (00).
B - 32	Index line text.
33	Null byte; indicates end of index entry.

After the last index entry, each card's data is stored. Card data is in one of four general formats: graphic and text, text only, graphic only, and blank. Blank cards consist of 4 null bytes; the other card formats are below:

Graphic & Text	Text Only	Graphic Only	
0 - 1	0 - 1#	0 - 1	Flag Determining whether or not the card contains an object.
2 - 5	*	2 - 5	Unique object ID.
6 - x	*	6 - x	The OLE object.
x+1 - x+2	*	x+1 - x+2	Character width, used for device independence.
x+3 - x+4	*	x+3 - x+4	Character height.
x+5 - x+C	*	x+5 - x+C	RECT: left - X-coordinate of the

```
                                                  upper-left corner.
                                          top   - Y-coordinate of the
                                                  upper-left corner.
                                          right - X-coordinate of the
                                                  lower-right corner.
                                          bottom- Y-coordinate of the
                                                  lower-right corner.
   x+D - x+E      *       x+D - x+E     Object type embedded=0, linked=1, or
                                        static=2 (values may change in the
                                        future).
   x+F - x+10   2 - 3     x+F - x+10#   Length of text entry.
   x+11 - y     4 - z         *         Text.
```

Note:

```
   x = 6 + size in bytes of the entire OLE object (the entire size of the
       object is not stored anywhere within the .CRD file). See below for
       more information on the OLE object size.
   y = x + 10 + length of text entry.
   z = 3 + length of text entry.
   # - These bytes are null if no object/text.
   * - These bytes do not exist if no object/text.
```

The first byte of any card's data entry is pointed to by bytes 6-9 in the index entry. Note that no null byte is used to indicate the end of the card's data entry; the next card's data entry immediately follows the last byte of the previous entry, which is null only if the previous card has no text (null length of text entry).

OLE Object

The size of the OLE object is not stored anywhere within the .CRD file. The OLE object could be loaded using OleLoadFromStream(); however, to get passed the OLE object, the file needs to be parsed. The OLE object's format description is documented in Appendix C of the "Object Linking and Embedding Programmer's Reference" version 1.0, published by Microsoft Press, and also in the Microsoft Windows Software Development Kit (SDK) "Programmer's Reference, Volume 1: Overview," Chapter 6, Object Storage Format. Below is an algorithm that uses the OLE object's format description to parse the OLE object in the .CRD file and pass it.

Need Five Primary Functions

Primary Function	Description
ReadLong()	- Reads a long from the file and advances the file pointer.
EatBytes(NumBytes)	- Reads and discards the specified number of bytes from the file and advances the file pointer.
RdChkVer()	- Reads the version number and advances the file pointer and returns TRUE if version is 1.0. To check the version number, the received value must be converted to Hex then checked against 0x0100. (See below for the algorithm of this function.)

```
   RdChkString()         - Reads the string and checks the value to see if
                           it is either METAFILEPICT, BITMAP, or DIB, then
                           returns TRUE; otherwise, returns FALSE. Advances
                           the file pointer too.
   SkipPresentationObj() - Reads and skips the variable-length presentation
                           object at the end of each object type.
                           (See below for the algorithm of this function.)
```

Algorithm to Skip Over the OLE Object

```
if (RdChkVer)                     // If the version is 1.0
  Format = ReadLong();            // 1==> Linked, 2==> Embedded, 3==> Static
  EatBytes(ReadLong());           // Class String
  if (Format == 3)                // Static object
    ReadLong();                   // Width in mmhimetric.
    ReadLong();                   // Height in mmhimetric.
    EatBytes(ReadLong());         // Presentation data size and data itself.
  else                            // Embedded or linked objects.
    EatBytes(ReadLong());         // Topic string.
    EatBytes(ReadLong());         // Item string.
    if (Format == 2)              // Embedded object.
      EatBytes(ReadLong());       // Native data and its size.
      SkipPresentationObj()       // Read and eat the presentation object.
    else                          // Linked object.
      EatBytes(ReadLong());       // Network name.
      ReadLong();                 // Network type and net driver version.
      ReadLong();                 // Link update options.
      SkipPresentationObj()       // Read and eat the presentation object.
```

SkipPresentationObj()

```
if (RdChkVer)                     // If the version is 1.0
  ReadLong();                     // Format ID
  if (RdChkString())              // if Class String is either
                                  // METAFILEPICT or BITMAP or DIB.
    ReadLong();                   // Width in mmhimetric.
    ReadLong();                   // Height in mmhimetric.
    EatBytes(ReadLong());         // Presentation data size and data itself
  else
    if (!ReadLong())              // if Clipboard format value is NULL
      EatBytes(ReadLong());       // Read Clipboard format name.
    EatBytes(ReadLong());         // Presentation data size and data itself.
```

RdChkVer()

```
OLEVer = ReadLong();
OLEVer = (((WORD)(LOBYTE(OLEVer))) << 8 | (WORD) HIBYTE(OLEVer);
if (OLEVer == 0x0100)              // Always use Hex value.
  return TRUE;
else
  return FALSE;
```

Additional references: 3.10
KBCategory:

KBSubCategory: UsrFmtRaremisc

INF: Windows 3.00 Sample Source Code for a Keyboard Filter

Article ID: Q66989

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

KBHOOK is a file in the Software/Data Library that contains sample code to demonstrate the installation and use of a system-wide keyboard filter function (otherwise known as a keyboard hook function). This code monitors the status of the CAPS LOCK key. Whenever the CAPS LOCK key is pressed, its status is displayed on the application's icon. Terminating the application removes the keyboard filter from the system.

KBHOOK can be found in the Software/Data Library by searching on the word KBHOOK, the Q number of this article, or S12795. KBHOOK was archived using the PKware file-compression utility.

More Information:

KBHOOK contains an application program, KEYAPP, and a dynamic-link library (DLL), KEYHOOK, that implements the keyboard filter function.

KEYAPP calls KEYHOOK to install the keyboard filter function and then makes itself iconic. Windows calls KEYHOOK each time a key is pressed. When the CAPS LOCK key is pressed, KEYHOOK posts a message to KEYAPP. KEYAPP processes the message by painting its icon to display the current state of CAPS LOCK. When KEYAPP is terminated, the KEYHOOK filter is removed from the system.

Because the keyboard filter function is called regardless of the application that is currently active, the code must be in memory at all times. When Windows is running in real mode with extended memory, code for inactive applications can be placed into extended memory banks that are switched out of memory. Windows's memory management scheme is defined so that code in fixed DLLs will remain in memory at all times and will remain available for execution.

For more information on the keyboard filter function, see the documentation for the SetWindowsHook() function on pages 4-419 through 4-427 in the "Microsoft Windows Software Development Kit Reference Volume 1." The "Microsoft Windows Software Development Kit Guide to Programming" provides additional information on DLLs.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrHksKeyboard

INF: Sample Code Demonstrates Windows 3.1 WH_MOUSE Hook
Article ID: Q81333

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

MOUSHOOK is a file in the Software/Data Library that demonstrates installing and using a WH_MOUSE mouse message hook. The mouse message hook in MOUSHOOK disables the right mouse button for all applications in the system by discarding WM_RBUTTONDOWN and WM_NCRBUTTONDOWN messages. The mouse hook function beeps each time it discards a message.

The MOUSHOOK file contains the source to an application program, TESTAPP, and a dynamic-link library (DLL), MOUSHOOK, that implements the mouse hook. The MOUSHOOK DLL uses the new hook functions provided by Windows 3.1: SetWindowsHookEx, UnhookWindowsHookEx, and CallNextHookEx.

TESTAPP calls MOUSHOOK to install the mouse hook. When TESTAPP is closed, it removes the mouse hook from the system.

MOUSHOOK can be found in the Software/Data Library by searching on the word MOUSHOOK, the Q number of this article, or S13297. MOUSHOOK was archived using the PKware file-compression utility.

Additional reference words: 3.10 softlib MOUSHOOK.ZIP
KBCategory:
KBSubcategory: UsrHksSetting

INF: Sample Code Demonstrates Using a WH_KEYBOARD Hook

Article ID: Q81334

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

KBHOOK2 is a file in the Software/Data Library that contains sample code to demonstrate installing and using a system-wide WH_KEYBOARD keyboard filter function (otherwise known as a keyboard hook function). This code monitors the status of the CAPS LOCK key. Whenever the CAPS LOCK key is pressed, its status is displayed on the application's icon. Terminating the application removes the keyboard filter from the system.

KBHOOK2 updates the KBHOOK sample to use new functions provided by Windows 3.1. For more information on the KBHOOK sample, query on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and kbhook

KBHOOK2 can be found in the Software/Data Library by searching on the word KBHOOK2, the Q number of this article, or S13291. KBHOOK2 was archived using the PKware file-compression utility.

More Information:

KBHOOK2 contains an application program, KEYAPP, and a dynamic-link library (DLL), KEYHOOK, that implements the keyboard filter function. The KEYHOOK DLL uses the new Windows 3.1 hook functions: SetWindowsHookEx, UnhookWindowsHookEx, and CallNextHookEx.

KEYAPP calls KEYHOOK to install the keyboard filter function and then makes itself iconic. Windows calls KEYHOOK each time a key is pressed. When the CAPS LOCK key is pressed, KEYHOOK posts a message to KEYAPP. KEYAPP processes the message by painting its icon to display the current state of CAPS LOCK. When KEYAPP is terminated, the KEYHOOK filter is removed from the system.

Additional reference words: 3.10 softlib KBHOOK2.ZIP
KBCategory:
KBSubcategory: UsrHksKeyboard

INF: Sample Code Uses Keyboard Hook to Access Help

Article ID: Q83233

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

F1CDHELP is a file in the Software/Data Library that demonstrates how an application can implement context sensitive help for the dialog boxes provided by the dynamic-link library (DLL) COMMDLG.DLL (the common dialog box DLL). When the user presses the F1 key, the application calls Windows Help.

The recommended user interface for an application to provide access to Windows Help is through the F1 key. This article describes a technique that can be used with the common dialog box DLL to bring up context sensitive help for each dialog box. The sample uses the SetWindowsHookEx function to set a task-specific keyboard hook, which monitors keyboard input and responds to the F1 key.

Because this application uses a task-specific hook, the hook function code resides in the application's EXE file and is not required to be in a fixed code page in a DLL. (The filter function for each system-wide hook must be in a DLL.)

F1CDHELP can be found in the Software/Data Library by searching on the word F1CDHELP, the Q number of this article, or S13267. F1CDHELP was archived using the PKware file-compression utility.

More Information:

The technique used in this article is straightforward and minimizes the number of global variables required. The application installs the keyboard hook and calls the RegisterWindowMessage function to define a help message. When an application registers the help message, the common dialog box DLL notifies the application each time the user chooses the Help button in one of the common dialog boxes. The DLL sends the help message to the window procedure of the dialog box's parent window.

When the keyboard hook function detects that the F1 key is pressed, it posts a WM_COMMAND message to the appropriate window procedure. In the F1CDHELP example, the message is posted either to the main window (when no dialog box is displayed) or to the dialog box's window procedure. If the message is posted to one of the common dialog boxes, wParam is set to pshHelp; the application simulates choosing the Help button in the common dialog box. Otherwise, wParam is set to IDM_HELPCONTENTS; the application simulates selecting a menu item in the application.

The following code demonstrates installing the hook and registering the help message in response to a WM_CREATE message:

```
   case WM_CREATE:
      // Install the keyboard hook

      lpfnKbrdHook = MakeProcInstance((FARPROC)KeyboardHook, ghInst);
      ghKbrdHook = SetWindowsHookEx(WH_KEYBOARD, lpfnKbrdHook,
                                    ghInst, GetCurrentTask());

      // Register the help message. The common dialog box DLL sends this
      // message when the user chooses the Help file in a common
      // dialog box.
      gwHelpMsg = RegisterWindowMessage((LPSTR)HELPMSGSTRING);
      break;
```

The following code demonstrates removing the keyboard hook in response to a WM_DESTROY message:

```
   case WM_DESTROY:
      UnhookWindowsHookEx(ghKbrdHook);
      break;
```

The hook function receives notification about the F1 key and posts a message as appropriate:

```
DWORD FAR PASCAL KeyboardHook(int iCode, WPARAM wParam, LPARAM lParam)
{
   if (iCode < 0 || iCode != HC_ACTION)
      return CallNextHookEx(ghKbrdHook, iCode, wParam, lParam);

   if (wParam == VK_F1)

      // If this is a repeat or the key is being released, ignore it.
      if (lParam & 0x80000000 || lParam & 0x40000000)
         return CallNextHookEx(ghKbrdHook, iCode, wParam, lParam);
      else
      {
         if (IsWindowEnabled(ghWnd)) // F1 pressed in main window?
            PostMessage(ghWnd, WM_COMMAND, IDM_HELPCONTENTS, 0L);
         else                        // F1 pressed in a dialog box
            PostMessage(GetActiveWindow(), WM_COMMAND, pshHelp, 0L);
      }

   return CallNextHookEx(ghKbrdHook, iCode, wParam, lParam);
}
```

The second PostMessage call above is executed when the user requests help in a specific common dialog box. This simulates choosing the Help button in the dialog box. Because the application registered the help message (during the processing of its WM_CREATE message), the common dialog box DLL will send the gwHelpMessage to the parent window procedure. An application can process this message as follows:

```
   default:
      if (message == gwHelpMsg)  // Help requested in a common dialog
                                 // box
      {
      // The lParam points to an OPENFILENAME or a CHOOSECOLOR data
```

```
      // structure. The application can differentiate between them by
      // checking the structure's size, which is in the first four
      // bytes (a DWORD) of the structure. This allows the application
      // to display different help for each of the common dialog boxes.

      dwStructSize = (DWORD)(*(LPDWORD)lParam);

      switch (dwStructSize)
         {
         case sizeof(OPENFILENAME):
            MessageBox((HWND)wParam, "Help requested for OpenFile",
                       gszAppName, MB_OK);
            break;

         case sizeof(CHOOSECOLOR):
            MessageBox((HWND)wParam, "Help requested for ChooseColor"
                       gszAppName, MB_OK);
            break;

         default:
            break;
         }
      break;
      }
      else // Not a help message
         return DefWindowProc(hWnd, message, wParam, lParam);
```

Additional reference words: 3.10 softlib F1CDHELP.ZIP
KBCategory:
KBSubcategory: UsrHks

INF: Cannot Alter Messages with WH_KEYBOARD Hook

Article ID: Q33690

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Problem:

I am writing a Windows hook that intercepts and translates the user's keystrokes so that the user can type characters contained in the upper half of our modified OEM character set. I have created a program that uses the WH_KEYBOARD hook function to intercept the user's keystrokes. I then translate the wParam parameter according to my look-up table and return the translated value via DefHookProc(). I am able to verify that my translation procedure works as expected; however, the application that has the focus is not receiving the modified character message.

I have changed the wParam value and then passed the hook code via DefHookProc(), but whatever application has the focus still receives the original character and not the translated character.

Response:

Keyboard messages cannot be altered with this hook. All that can be done is to "swallow" the message (return TRUE) or have the message passed on (FALSE). In a keyboard hook function, when you return DefHookProc(), you are passing the event to the next hook procedure in the potential hook chain, and giving it a chance to look at the event to decide whether or not to discard it. You are not passing the message to the system as if you had called DefWindowProc() from a Window procedure.

To change the value of wParam (and hence the character message that is received by the window with the focus), you must install the WM_GETMESSAGE and WH_CALLWNDPROC hooks. The WH_GETMESSAGE hook traps all messages retrieved via GetMessage() or PeekMessage(). This is the way actual keyboard events are received: the message is placed in the queue by Windows and the application retrieves it via GetMessage() or PeekMessage(). However, because applications can send keyboard messages with SendMessage(), it is best to also install the WH_CALLWNDPROC hook. This hook traps messages sent to a window via SendMessage().

These hooks pass you the address of the message structure so you can change it. When you return DefHookProc() within a WH_GETMESSAGE or WH_CALLWNDPROC hook procedure, you are passing the address of the (potentially altered) contents of the message structure on to the next hook function in the chain. If you alter the wParam before passing on the message, this will change the character message eventually received by the application with the focus.

Keep in mind that the hook callback procedure must be placed in a DLL with fixed code so that it will be below the EMS line and thus will always be present. If the hook callback procedure is not in a fixed code segment, it could be banked out when it is called, and this would crash the system.

Additional reference words: 2.03 2.x 3.00
KBCategory:
KBSubcategory: UsrHksKeyboard

INF: Correction to JournalRecordProc Documentation
Article ID: Q86007

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In the documentation for the JournalRecordProc on page 561 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" manual and in the Windows SDK online help file, the lParam parameter passed to a JournalRecordProc callback function is documented incorrectly.

The documentation should state the lParam points to an EVENTMSG data structure, which is defined as follows:

```
   typedef struct tagEVENTMSG {
       UINT  message;
       UINT  paramL;
       UINT  paramH;
       DWORD time;
   } EVENTMSG;
```

The EVENTMSG data structure is defined in the WINDOWS.H include file provided with version 3.1 of the Windows SDK.

Additional reference words: 3.10 WH_JOURNALRECORD on-line
KBCategory:
KBSubcategory: UsrHksJournal

INF: Windows Journal Hooks Sample Source Code

Article ID: Q37138

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

JOURNAL is a file in the Software/Data Library that contains sample code to demonstrate the use of the journal hooks WH_JOURNALRECORD and WH_JOURNALPLAYBACK. The JOURNAL archive file contains the source to APP.EXE; WNDPROC.C is one of the source files.

The APP.EXE application serves only one purpose; it illustrates how to call the dynamic-link library (DLL) that performs all of the journaling work. The WNDPROC.C application module contains calls to the DLL to record and play back messages. One of the source modules for the DLL, JOURNAL.C, contains all the code required to create DIARY.BIN, a file of messages. The messages in DIARY.BIN can be played back at a later time. The DIARY.BIN file is overwritten each time the user chooses Record On from the menu.

The JOURNAL code can serve as the basis of a "hands off" application demonstration. The application demonstrating itself can call the high-level functions in the DLL to play each event.

The DIARY.BIN file is also included in the JOURNAL archive to demonstrate using journal hooks. To demonstrate playing the contents of a DIARY.BIN file, select Play from the menu.

JOURNAL can be found in the Software/Data Library by searching on the word JOURNAL, the Q number of this article, or S10062. JOURNAL was archived using the PKware file-compression utility.

Additional reference words: 1.04 2.03 2.10 3.00 2.x softlib
JOURNAL.ZIP SetWindowsHook
KBCategory:
KBSubcategory: UsrHksJournal

INF: Importance of Calling DefHookProc()

Article ID: Q74547

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When an application installs a hook using SetWindowsHook, Windows adds the hook's callback filter function to the hook chain. It is the responsibility of each callback function to call the next function in the chain. DefHookProc() is used to call the next function in the hook chain.

More Information:

If a callback function does not call DefHookProc(), none of the filter functions that were installed before the current filter will be called. Windows will try to process the message and this could hang the system.

Only a keyboard hook (WH_KEYBOARD) can suppress a keyboard event by not calling DefHookProc and returning a 1. When the system gets a value of 1 from a keyboard hook callback function, it discards the message.

Furthermore, when the hook callback function receives a negative value for the nCode parameter, it should pass the message and the parameters to DefHookProc() without further processing. When nCode is negative, Windows is in the process of removing a hook callback function from the hook chain.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrHksForwarding

PRB: DLL System Hook Function Not Affecting Apps System-Wide

Article ID: Q88190

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

SYMPTOMS

When you are trying to install a system hook function with SetWindowsHookEx in a dynamic-link library (DLL), the hook function is executed only within the calling application even though it is supposed to affect all applications system-wide.

CAUSE

In Windows 3.0, the minimum required compile option to generate the appropriate entry/exit code sequence for a Windows application is the /Gw switch.

With Microsoft C/C++ version 7.0, however, the documentation states that the /Gw and /GW switches should be used only for applications that must run in real mode Windows. Because real mode is no longer available in Windows 3.1, most programs should now be built using the /GA switch (/GD for DLLs). The C 7.0 /GA and /GD switches require that exported functions (especially callback functions) be explicitly marked as __export if the switch is to affect them.

RESOLUTION

If the DLL containing the hook function is compiled with the /GD switch, the hook function should be explicitly marked with the __export keyword:

```
LRESULT FAR PASCAL __export CbtFunc (int Code,
                                     WPARAM wParam, LPARAM lParam);
```

Additional reference words:3.10
KBCategory:
KBSubcategory: UsrHks

INF: How to Stop a Journal Playback
Article ID: Q98486

Summary:

To stop a "journal playback" when a specified key is pressed, the filter function must determine whether the key was pressed and then call the UnhookWindowsHookEx function to remove the WH_JOURNALPLAYBACK hook.

More Information:

To determine the state of a specified key, the filter function must call the GetAsyncKeyState function when the nCode parameter equals HC_SKIP. The HC_SKIP hook code notifies the WH_JOURNALPLAYBACK filter function that Windows is done processing the current event.

The GetAsyncKeyState function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to the GetAsyncKeyState function. If the most significant bit of the return value is set, the key is down; if the least significant bit is set, the key was pressed after a preceding GetAsyncKeyState call.

If the filter function calls the GetAsyncKeyState function after the specified key was pressed and released, then the most significant bit will not be set to reflect a key-down. Thus, a test to check whether the specified key is down fails. Therefore, the least significant bit of the return value must be checked to determine whether the specified key was pressed after a preceding call to GetAsyncKeyState function. Using this technique of checking the least significant bit requires a call to the GetAsyncKeyState function before setting the WH_JOURNALPLAYBACK hook. For example:

```
// When setting the journal playback hook.
      .
      .
      .
      // Reset the lease significant bit.
      GetAsyncKeyState( VK_CANCEL );

      // Set a system-wide journal playback hook.
      g_hJP = SetWindowsHookEx( WH_JOURNALPLAYBACK,
                        FilterFunc,
                        g_hInstDLLModule,
                        NULL );
      .
      .
      .

// Inside the filter function
      .
      .
      .
      if ( nCode == HC_SKIP )
          if ( GetAsyncKeyState( VK_CANCEL) )
```

```
              UnhookWindowsHookEx( g_hJP );
      .
      .
      .
```

Additional reference words: 3.x 3.00 3.10
KBCategory:
KBSubcategory: UsrHksJournal

INF: Calling SendMessage() Inside a Hook Filter Function

Article ID: Q74857

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A hook filter function should not call SendMessage() to pass intertask messages because this behavior can create a deadlock condition in Windows. If a hook filter function is called as a result of an intertask SendMessage(), and if the hook function then yields control with an intertask SendMessage(), a message deadlock condition may occur. For this reason, hook filters should use PostMessage() rather than SendMessage to pass messages to other applications.

Note that a hook filter can use SendMessage() to pass a message to the current task because this will not yield the control.

Section 1.1.5 of the "Microsoft Windows Software Development Kit Reference Volume 1" has more information on message deadlocks.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrHksRare/misc

PRB: SetWindowsHookEx() Fails to Install Task-Specific Filter

Article ID: Q92659

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

SYMPTOMS

In Windows version 3.1, the SetWindowsHookEx() function fails when it is called to install a task-specific filter (hook) that resides in a DLL.

CAUSE

According to the documentation, the third parameter to the SetWindowsHookEx() function must be the instance handle of the application or the DLL that contains the filter function. However, because of a problem in Windows 3.1, the SetWindowsHookEx() function fails when it is called to install a task-specific filter using the DLL's instance handle.

Note that such a problem does not exist when the SetWindowsHookEx() function is called to install a system-wide filter in a DLL. The DLL's instance handle is accepted as a valid parameter. The first argument passed to the LibMain function of a DLL contains its instance handle.

RESOLUTION

To install a task-specific filter that resides in a DLL, pass the module handle of the DLL as the third parameter to the SetWindowsHookEx() function. The module handle can be retrieved using the GetModuleHandle() function. For example, to install a task-specific keyboard filter, the code might resemble the following:

```
g_hHook = SetWindowsHookEx( WH_KEYBOARD,
                   HookCallbackProc,
                   GetModuleHandle( "HOOK.DLL" ),
                   hTargetTask );
```

This resolution is compatible with future versions of Windows.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrHksSetting

INF: Sample Code for WH_CALLWNDPROC and WH_GETMESSAGE Hooks
Article ID: Q76588

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

HOOKALL is a sample program in the Software/Data Library that demonstrates the use of WH_GETMESSAGE and WH_CALLWNDPROC hooks. The sample contains the code for a dynamic-link library (DLL) with functions to set and remove the hooks, as well as the hook filter functions. The filter functions use OutputDebugString to display a message when they receive a WM_LBUTTONDOWN message. The sample also contains code for an application that tests the DLL.

HOOKALL can be found in the Software/Data Library by searching on the word HOOKALL, the Q number of this article, or S13178. HOOKALL was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrHksGetmess

INF: Message Structure Used by WH_CALLWNDPROC Filter Function

Article ID: Q76589

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The message structure passed to the WH_CALLWNDPROC filter function is documented incorrectly on pages 4-120 and 4-121 of the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0. The correct structure is as follows:

```
struct {
   LONG lParam;
   WORD wParam;
   WORD wMsg;
   WORD hWnd;
   }
```

For more information on the WH_CALLWNDPROC filter function, query on the words:

prod(winsdk) and wh_callwndproc

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrHksCallwndpr

PRB: Using ToAscii() in Journal Record Hooks

Article ID: Q99337

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1

SYMPTOMS
========

When a journal record hook procedure calls the ToAscii() function, dead keys are apparently no longer processed correctly due to a side effect.

CAUSE
=====

The current implementation of ToAscii() determines whether the key that is currently being processed is a dead key; if yes, it collapses the next keystroke seen in the input stream with the dead key for Windows to display. This assumption is being made for Windows, which calls the ToAscii() function during normal processing of keystrokes. Calling ToAscii() in a journal record hook clears the flag that indicates the presence of a dead key; thus, the key combination will not be displayed correctly on the screen although it will appear correctly when the recorded key strokes are played back.

RESOLUTION
==========

The journal record hook procedure should call the ToAscii() function twice with the same parameters when processing a dead key [that is, the call to ToAscii() returns -1]. This will reset the dead key flag internally.

Note that the source code for the ToAscii() function is publicly available because it resides in the keyboard device driver whose source is shipped with the Windows Device Development Kit (DDK).

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrHksJournal

INF: Determining Message Removal from WH_GETMESSAGE Hook
Article ID: Q104068

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows,
   version 3.1

A WH_GETMESSAGE hook is called each time PeekMessage or GetMessage is called. The wParam parameter of the hook indicates whether the message is being removed from the queue or merely being looked at.

If PeekMessage is called with the PM_NOREMOVE flag, the message will not be removed from the queue. A WH_GETMESSAGE hook is called each time PeekMessage or GetMessage is called, and therefore the hook may be called more than once for the same message if PeekMessage is called with PM_NOREMOVE.

The hook can determine whether the message is being removed by using the information in the wParam parameter. wParam contains the PM_* flags that were used in the PeekMessage call. GetMessage calls PeekMessage with the PM_REMOVE flag. The Windows 3.1 SDK does not document the value of wParam. It will be documented in future versions.

Note that PM_NOREMOVE is defined as 0x0000, and therefore must be used carefully in any Boolean conditional code. Code similar to the following may be used in the hook:

```
   if (wParam & PM_REMOVE)
       Message is being removed;
   else Message is being looked at but not being removed. The hook
       may be called again for the same message.
```

Additional reference words: 3.00 3.10 multiple
KBCategory:
KBSubcategory: UsrHksGetmess

INF: Using Quoted Strings with Profile String Functions

Article ID: Q69752

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Microsoft Windows provides profile files which are a mechanism for an application to store configuration about itself. The WIN.INI file is the system profile file in which Windows stores configuration information about itself. In versions of Windows prior to version 3.0, applications also stored configuration information in the WIN.INI file. Windows 3.0 introduced private profile files, which can store application-specific information.

An application can retrieve information from a profile file by calling the GetProfileString or GetPrivateProfileString function. If the profile file associates the specified lpKeyName value with a string that is delimited by quotation marks, Windows discards the quotation marks when it copies the associated string into the application-provided buffer.

For example, if the following entry appears in the profile file:

```
   [application name]           [application name]
   keyname = 'string'     or     keyname = "string"
```

The GetPrivateProfileString and GetProfileString functions read the string value and discard the quotation marks.

More Information:

This behavior allows spaces to be put into a string. For example, the profile entry

```
   keyname = string
```

returns the string without a leading space, whereas

```
   keyname = ' string'     or     keyname = " string"
```

returns the string with a leading space.

Doubling quotation marks includes quotation marks in the string. For example:

```
   keyname = ''string''    or     keyname = ""string""
```

returns the string with its quotation marks -- 'string' or "string".

Additional reference words: 3.00 3.10 SR# G910131-36 MICS3 R3.9

KBCategory:
KBSubcategory: UsrIniPrivprof

INF: Using WriteProfileString to Delete WIN.INI Entries

Article ID: Q65111

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The WriteProfileString function provides a mechanism to maintain the system profile file WIN.INI. In versions of Windows prior to version 3.0, an application could call the WriteProfileString function to add entries to WIN.INI or to modify existing entries. However, to delete an entry from WIN.INI, the user was required to edit the file manually.

The WriteProfileString function was extended in Windows 3.0 to enable an application to delete any of the following from WIN.INI:

1. A value associated with a key name.
2. A line that contains a particular key name.
3. An entire section.

This article details the three deletion scenarios listed above.

More Information:

The parameters passed to the WriteProfileString function refer to the following selection from an example WIN.INI file:

```
[MY_APP]
KeyName = DATA
```

To delete the line KeyName = DATA, call the WriteProfileString function as follows:

```
WriteProfileString("MY_APP", "KeyName", NULL);
```

To delete the value DATA, change the call to the following:

```
WriteProfileString("MY_APP", "KeyName", "");
```

To delete the entire MY_APP section, make the following call:

```
WriteProfileString("MY_APP", NULL, NULL);
```

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrIniWinini

INF: WritePrivateProfileString Documented Incorrectly
Article ID: Q78419

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The documentation for the WritePrivateProfileString function on pages 4-462 through 4-464 of the "Microsoft Windows Software Development Kit Reference Volume 1" includes the following statement in the description of the lpFileName parameter:

> The WritePrivateProfileString does not create a file if lpFileName contains the fully qualified pathname of a file that does not exist.

This statement is incorrect. The WritePrivateProfileString function does create the file under these conditions.

Microsoft has confirmed this to be an error in version 3.0 of the Windows Software Development Kit (SDK) documentation.

This error has been corrected on Page 996 of the "Microsoft Windows Version 3.1 Software Development Kit Programmer's Reference, Volume 2: Functions".

Additional reference words: 3.00 docerr
KBCategory:
KBSubcategory: UsrIniPrivprof

INF: Control Panel Doesn't Respond to WM_WININICHANGE Messages

Article ID: Q68360

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Control Panel application does not respond to WM_WININICHANGE messages under Windows version 3.0. Microsoft considered including this functionality; however, it was decided that, because the Control Panel changes various items in the system configuration file (WIN.INI), responding to a WM_WININICHANGE message could cause a conflict. (For example, if another application changes the text color while the Control Panel has its color dialog displayed, what is the proper action for the Control Panel?)

To work around this potential problem, the Control Panel reinitializes itself each time a new dialog box is displayed. However, this initialization does not involve querying the WIN.INI file with GetProfileString().

More Information:

Windows caches the WIN.INI file. Therefore, if a regular text editor (other than Notepad) is used to modify the file, the disk file is updated, however, the cache is not.

When a WM_WININICHANGE message is broadcast to all top-level windows, each application can check the WIN.INI cache for relevant changes. However, applications will not look at the WIN.INI file on disk.

One way to update the cache is to exit Windows and restart. Another way to update the cache is to call WriteProfileString(NULL,NULL,NULL), which forces a flush of the cache. A subsequent call to GetProfileString() will cause the WIN.INI file on disk to be read into the cache.

However, the changes will not be reflected by the Control Panel unless the user exits and re-enters Windows. Reflecting changes to WIN.INI in the Control Panel without first exiting Windows is under consideration for inclusion in a future release of Windows.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrIniWinini

INF: Guide to Programming Get Printer Information Code Wrong

Article ID: Q68807

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The sample code in section 12.2 of the "Microsoft Windows Software Development Kit Guide to Programming" is incorrect. On page 12-3, the code by bullet 1 that reads

```
        GetProfileString("windows",
                         "device",
                         pPrintInfo,
                         (LPSTR) NULL, 80);
```

must be changed to read:

```
        GetProfileString("windows",
                         "device",
                         "",
                         pPrintInfo, 80);
```


More Information:

The fourth parameter to GetProfileString() specifies a buffer to receive the result of the GetProfileString() operation. In the sample above, this value should be pPrintInfo, not NULL.

The third parameter to GetProfileString() is the value to return if the specified key does not exist in the initialization file. This value must not be the NULL pointer. However, a pointer to an empty string ("") is perfectly acceptable and will return an empty string if the specified key does not exist.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrIniWinini

Fatal Exit Code 0x0506 Definition

Article ID: Q69888

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Fatal Exit code 0x0506 is not documented in Appendix C of the "Microsoft Windows Software Development Kit Reference Volume 2" version 3.0.

This Fatal Exit will occur if the GetProfileString() or GetPrivateProfileString() function is called with the lpDefault parameter set to NULL. lpDefault specifies a string to return in the lpReturnedString buffer if no match for lpKeyName is found.

To correct this Fatal Exit, always provide a valid string pointer for the lpDefault parameter.

GetProfileString() and GetPrivateProfileString() are documented on pages 4-202 and 4-199, respectively, in volume 1 of the SDK reference.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrIniPrivprof

FIX: Possible Cause of Cached Profile File Corruption

Article ID: Q70801

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9103009

SYMPTOMS

If the lpString parameter to the WritePrivateProfileString function points to a null string, when the function is called, the application will experience an unrecoverable application error (UAE). The lpFileName private profile and any other profile cached in memory are deleted.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrIniPrivprof

INF: Windows Documentation of WIN.INI Definitions

Article ID: Q35983

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Windows version 3.0 Setup program copies two text files, WININI.TXT and WININI2.TXT, which describe WIN.INI, and three text files called SYSINI.TXT, SYSINI2.TXT, and SYSINI3.TXT, which describe SYSTEM.INI. Additional information can be found in the "Microsoft Windows User's Guide" shipped with the version 3.0 retail Windows product.

For version 2.0, documentation that defines the various items in the WIN.INI file can be found in Chapter 7, section 7.4, "Windows Initialization File," in the "Microsoft Windows Software Development Kit Programmer's Reference," starting on page 653.

Additional information on WIN.INI is included on page 271 of the "Microsoft Windows User's Guide" that is shipped with version 2.0 of the retail Windows product.

Additional reference words: 2.03 2.10 3.00 2.x debug options
KBCategory:
KBSubcategory: UsrIniWinini

INF: Private Profile (INI) Files Not Designed as Database

Article ID: Q74602

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows graphical environment, a private profile or initialization file, as the name suggests, is designed to be used when a program initializes and terminates. A profile stores a limited amount of information from one program session to the next. A profile is are not designed to serve as a database.

More Information:

Windows assumes that an INI file is not larger than 64K; file access is not guaranteed beyond that point. Even if a text editor is used to extend the file, Windows does not search for information past the 64K boundary.

Also, Windows performs a linear search of INI files; therefore, the longer the file becomes, the longer it takes to access an item at the end of the file. Each time an application accesses an INI file, Windows opens and closes the file which incurs additional overhead.

One alternative to using initialization files to store large amounts of information is for the program to write to and maintain its own files. This approach is faster, more flexible, and more reliable than using profile for purposes for which it is not designed.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrIniPrivprof

INF: When to Use WIN.INI or a Private INI File

Article ID: Q74608

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application should use a private profile (INI) file to store initialization information where possible rather than the main INI file in Windows, WIN.INI. Profile files are not designed for use as a database or a mass-storage device.

More Information:

Applications use the profile functions in the Windows application programming interface (API) to save and retrieve initialization settings. The following profile functions are used with WIN.INI:

```
GetProfileInt
GetProfileString
WriteProfileString
```

Until Windows version 3.0, applications stored their initialization data in only one global place (WIN.INI). Windows version 3.0 added a complimentary set of functions to the Windows API to enable an application to store its initialization data in a private INI file. These functions are as follows:

```
GetPrivateProfileInt
GetPrivateProfileString
WritePrivateProfileString
```

The following factors provided the motivation for the addition of private INI files:

- INI files are limited to 64K in size.

- Windows ignores the portion of INI files past 64K. Therefore, if enough applications use WIN.INI rather than separate, private INI files, some of the user's INI data may be ignored.

- No consistent way exists for users to remove old, unneeded information from the WIN.INI file. Typically, when an application is removed from the system, the files are deleted from the application's directory. However, the corresponding information may not be deleted from WIN.INI. Alternately, if initialization data is stored in a private INI file in the application's directory or in a file with the application's name, the user is much more likely to delete the obsolete information.

- Windows uses a linear search to find information in INI files.

Therefore, smaller INI files provide faster performance.

By default, INI files are created in the Windows directory. However, an application should always use a fully qualified path to a different directory because the Windows system directory is a shared resource in a Windows network setup.

Do not use the private profile functions with the WIN.INI file. Windows caches a copy of WIN.INI and one private INI file. This caching scheme may be confused if WIN.INI is altered using the private profile functions.

Applications should use INI files conservatively. Use as few sections and as few lines as possible. For example, do not save the coordinates of a window individually, as follows:

```
[window save pos]
ul = 10
ur = 10
ll = 100
lr = 100
```

Instead, use one line, as follows:

```
[save_pos]
window=10 10 100 100
```

This is a more efficient use of space and is much faster.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrIniWinini

INF: Program Manager Restrictions Settings
Article ID: Q75337

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

In Windows version 3.1, a new section will be added to the PROGMAN.INI initialization file to allow system and network administrators to restrict the functionality of the Program Manager to make it better suited to handle shared files.

There are five new options available to prevent a user from modifying his group's configuration or from running untrusted software. These options appear in the section labeled [Restrictions]. This article details each of the options below.

More Information:

```
[Restrictions]
NoRun=[0/1]
```

When the NoRun switch is set to 1, the Run command on the File menu is disabled. The default value of this option is 0 (allow File Run).

```
[Restrictions]
NoClose=[0/1]
```

When the NoClose option is set to 1, this option prevents the user from exiting the Program Manager, through the File menu, the System menu, the ALT+F4 accelerator or the Task Manager. The default value of this option is 0 (allow exit).

```
[Restrictions]
NoSaveSettings=[0/1]
```

When the NoSaveSettings option is set to 1, this option prevents the user from saving the main window position and the load order of the groups. It also disables the "Save Settings on Exit" command on the Options menu. The default value of this option is 0 (allow the user to save settings).

```
[Restrictions]
NoFileMenu=[0/1]
```

When the NoFileMenu option is set to 1, this option disables the File menu entirely. All commands on this menu are disabled as well. The default value of this option is 0 (the File menu is enabled).

```
[Restrictions]
```

EditLevel=<value>

This EditLevel option controls the extend to which a user can modify read/write groups (shared, read-only groups may never be modified). The following values are recognized:

0 Allow any change (default)
1 User cannot create, delete, or rename groups
2 Value 1 restrictions; also, user cannot create or delete items
3 Value 2 restrictions; also, user cannot change item command lines
4 Value 3 restrictions; also, user cannot change any item property

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrIniRare/misc

INF: INIHEADR Sample Reads Section Headers from .INI Files

Article ID: Q104096

The information in this article applies to:

 - Microsoft Windows Software Development Kit (SDK) for Windows
   versions 3.0 and 3.1

SUMMARY
=======

The Windows Software Development Kit (SDK) does not define a function to read all of the section headers from a profile (.INI) file. An application must read the entire .INI file and parse it to obtain the section headers.

NOTE: Microsoft recommends using Windows's application programming interfaces (APIs) to read .INI files. The method described below is not guaranteed to work on future releases, especially for system .INI files such as WINFILE.INI, WIN.INI, SYSTEM.INI, CONTROL.INI, and so forth.

The INIHEADR sample defines a function called GetPrivateProfileSections(), which parses a given .INI file and returns a buffer containing the section headers separated by NULLs and terminated by a double NULL. SECTION.C contains the function and a necessary helper function. The sample allows the user to choose an .INI file using the Open common dialog box. The section headings are then placed in a list box.

INIHEADR can be found in the Software/Data Library by searching on the word INIHEADR, the Q number of this article, or S14294. INIHEADR was archived using the PKware file-compression utility.

MORE INFORMATION
================

A profile file (WIN.INI) or a private profile (such as CLOCK.INI) must have the following form:

```
   [section heading]
   entry=value
      .
      .
      .
```

For example, given a profile containing the following section headings

```
   [windows]
      .
      .
      .
   [Desktop]
```

```
       .
       .
       .
   [Extensions]
```

GetPrivateProfileSections() parses the profile file and places the following in a buffer:

```
   windows<NULL>Desktop<NULL>Extensions<NULL><NULL>
```

Additional reference words: 3.00 3.10 softlib iniheadr.zip
KBCategory:
KBSubcategory: UsrIni

INF: Return Value of SwapMouseButton() Documented Incorrectly
Article ID: Q71310

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Page 4-437 of the "Windows Software Development Kit Reference Volume 1" documents the return value for the SwapMouseButton() function incorrectly. The return value is documented as follows:

> The return value specifies the outcome of the function. It is TRUE if the function reversed the meaning of the mouse buttons. Otherwise, it is FALSE.

Instead, the documentation should read:

> The return value specifies the meaning of the mouse buttons immediately prior to the SwapMouseButton call. It is TRUE if the meaning of the mouse buttons was reversed. Otherwise, it is FALSE.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrInpRare/misc

INF: Changing the Mouse Cursor for a Window

Article ID: Q31747

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

There are two different methods that an application can use to change the mouse cursor in a window:

- The application can use the SetCursor function, which will change the mouse cursor immediately.

- The application can use the SetClassWord function to change the mouse cursor for all windows of a particular window class. This method affects the mouse cursor only while it resides within the client area of a window of that class.

This article provides additional details regarding these two methods.

More Information:

A number of factors should be considered during the design of an application that changes the mouse cursor. The major consideration is that Windows sends the WM_SETCURSOR message any time the mouse cursor moves on the screen. Normally, Windows sends the message to the window "under" the mouse cursor. However, if a window sets the mouse capture, using the SetCapture function, that window receives all mouse messages, without regard to the position of the mouse cursor.

When an application calls SetCursor, the mouse cursor changes to reflect the cursor specified in the call. The cursor retains that shape until SetCursor is called again, either explicitly by the application, by the DefWindowProc function, or by another application.

Because Windows is a nonpreemptive multitasking environment, no other application will gain control of the processor until the application that has the processor releases it. If the application calls one of a number of Windows functions, it can potentially lose control of the processor. For a list of Windows functions that can cause control of the processor to pass between applications, search on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and nonpreemptive and multitasking

When the DefWindowProc or DefDlgProc function processes a WM_SETCURSOR message, it calls SetCursor to change the cursor to the default cursor for the application's window.

The application can prevent the cursor from changing by processing the WM_SETCURSOR message. A typical application that processes

WM_SETCURSOR will have a global variable for the handle to the current cursor. When the application receives a WM_SETCURSOR message, it checks the global variable. If the variable is NULL, the application passes the WM_SETCURSOR message to DefWindowProc. Otherwise, the application calls SetCursor with the value in the global variable. To return the cursor to the window default cursor, set the global variable to NULL.

When Windows sends a WM_SETCURSOR message, it places the hit-test area code into the low-order word of the lParam parameter. The application can use the hit-test area code to determine what particular portion of the window is "under" the mouse cursor. For more information on the hit-test area codes, see the documentation for the WM_NCHITTEST message in the "Microsoft Windows Software Development Kit Reference Volume 1."

Additional reference words: 2.03 2.10 3.00 3.10 RegisterClass 2.x
KBCategory:
KBSubcategory: UsrInpRare/misc

PRB: Input Focus Lost When Control Returns From Windows Help

Article ID: Q85896

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

SYMPTOMS

When an application calls the WinHelp function and specifies HELP_CONTEXTPOPUP as the value for the fuCommand parameter, the input focus disappears. When the user presses the ALT key, the system menu appears for a hidden copy of the Windows Help application.

CAUSE

The Windows Help application does not properly restore the focus to the calling application when the pop-up window is destroyed.

RESOLUTION

Microsoft has confirmed this to be a problem with Windows Help version 3.1. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrInpFocus

INF: Detecting Keystrokes While a Menu Is Pulled Down

Article ID: Q35930

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In the Windows environment, there are two methods that an application can use to receive notification that a key is pressed while a menu is dropped down.

The easier method is to process the WM_MENUCHAR message that is sent to an application when the user presses a key that does not correspond to any of the accelerator keys defined for the current menu.

The other method is to use a message filter hook specified with the SetWindowsHook function. The hook function can process a message before it is dispatched to a dialog box, message box, or menu.

The hook functions and the WM_MENUCHAR message are documented in the "Microsoft Windows Software Development Kit Reference--Volume 1."

Additional reference words: 2.x 3.00
KBCategory:
KBSubcategory: UsrInpRare/misc

PRWIN9105003: FatalExit() Interacts Through Terminal Only
Article ID: Q72497

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9105003

SYMPTOMS
Under the debugging version of Windows, when a debugger is not in use and the FatalExit() function interacts with the user, the interaction is through a terminal connected to the AUX port and not through the console keyboard and a secondary monitor if one is installed.

STATUS
In versions of Windows earlier than version 3.0, and in Windows 3.0 real mode, the device driver OX.SYS can be installed to redirect information from FatalExit() to a secondary monitor and the console keyboard. Under Windows 3.0 standard and enhanced modes, OX.SYS can be used to redirect output; however, it will not process input from the console keyboard. After a FatalExit() call has occurred, the computer must be rebooted to regain control.

Microsoft has confirmed this to be a limitation of the Windows Software Development Kit (SDK) version 3.0. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrInpRare/misc

PRB: WM_CHARTOITEM Messages Not Recieved by Parent of List Box

Article ID: Q72552

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS
When keyboard input is sent to a list box that has the LBS_WANTKEYBOARDINPUT style bit set, its parent does not receive WM_CHARTOITEM messages; however, WM_VKEYTOITEM messages are received.

CAUSE
The list box has the LBS_HASSTRINGS style bit set.

RESOLUTION
Windows sets the LBS_HASSTRINGS style bit for all list boxes except owner-draw list boxes. An owner-draw list box can be created with this style bit turned on or off. For owner-draw list boxes, the state of the LBS_HASSTRINGS style bit determines which messages are sent. WM_CHARTOITEM messages and WM_VKEYTOITEM messages are mutually exclusive.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrInpRare/misc

PRWIN9105004: MapVirtualKey() Maps Keypad Keys Incorrectly
Article ID: Q72583

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: PRWIN9105004

SYMPTOMS
   When MapVirtualKey() is passed valid virtual key codes for the keys
   on the numeric keypad, invalid scan codes and ASCII codes are
   returned.

CAUSE
   The Windows keyboard driver does not map all valid virtual keys to
   their scan codes and ASCII equivalents.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. We are researching this problem and will post new information
   here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrInpKeysapi

INF: Differentiating Between the Two ENTER Keys

Article ID: Q77550

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application may find it useful to differentiate between the user pressing the ENTER key on the standard keyboard and the ENTER key on the numeric keypad. Either action creates a WM_KEYDOWN message and a WM_KEYUP message with wParam set to the virtual key code VK_RETURN. When the application passes these messages to TranslateMessage, the application receives a WM_CHAR message with wParam set to the corresponding ASCII code 13.

To differentiate between the two ENTER keys, test bit 24 of lParam sent with the three messages listed above. Bit 24 is set to 1 if the key is an extended key; otherwise, bit 24 is set to 0 (zero). The contents of lParam for these messages is documented in the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0 of the SDK and in the SDK Reference Volume 3, "Messages, Structures, and Macros."

Because the keys in the numeric keypad (along with the function keys) are extended keys, pressing ENTER on the numeric keypad results in bit 24 of lParam being set, while pressing the ENTER key on the standard keyboard results in bit 24 clear.

The following code sample demonstrates differentiating between these two ENTER keys:

```
   case WM_KEYDOWN:
      if (wParam == VK_RETURN)    // ENTER pressed
         if (lParam & 0x1000000L) // Test bit 24 of lParam
            {
            // ENTER on numeric keypad
            }
         else
            {
            // ENTER on the standard keyboard
            }
      break;
```

Additional reference words: 3.00 SR# G911007-58 return 3.10
KBCategory:
KBSubcategory: UsrInpKeysapi

INF: Installed Language, Character Set, and Code Page

Article ID: Q80947

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Windows 3.0 supports a number of national languages, character sets, and code pages. Windows 3.0 does not, however, contain a function to provide information about what language, character set, or code page is installed. An application developed for the Windows environment can read the contents of the WIN.INI and SYSTEM.INI files to determine what support is installed. This article details how to obtain this information.

More Information:

Current language information is available in the SYSTEM.INI file. In the [keyboard] section, if the oemansi.bin= line has an entry in the form "xlatXXX.bin," XXX is the number of the installed code page. The Latin 2 character set corresponds to code page 852.

The installed national language support is also available in the SYSTEM.INI file. In the [boot] section, if the language.dll= line has an entry in the form "langXXX.dll," XXX is the three-letter abbreviation for the name of the country.

The three-letter abbreviations are listed in the [language] and [codepages] section of the SETUP.INF file. The Windows Setup program copies SETUP.INF into the Windows system directory (by default, C:\WINDOWS\SYSTEM).

Note: Windows 3.0 does not update the sLanguage field in the [intl] section of the WIN.INI file when support for the Eastern European (Latin 2) character set is installed.

For additional information, see "Adapt Your Program for Worldwide Use with Windows Internationalization Support" starting on page 29 of the Nov-Dec 1991 issue (volume 6, number 6) of the "Microsoft Systems Journal."

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrLoc

INF: Writing International Applications for Windows 3.00
Article ID: Q65124

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:
Note: This article is part of a set of seven articles, collectively called the "Windows 3.00 Developer's Notes." More information about the contents of the other articles, and procedures for ordering a hard-copy set, can be found in the knowledge base article titled "The Windows 3.00 Developer's Notes" (Q65260).

This article can be found in the Software/Data Library by searching on the keyword INTLAPPS or S12683.

More Information:

Microsoft(R) Windows(TM) version 3.00 provides an environment that allows you to give your applications country and language independence.

This document is a collection of information related to international support in Windows 3.00. For more information about functions mentioned in this document, see the documentation included with the Windows 3.00 Software Development Kit (SDK) and Device Development Kit (DDK).

==
CREATING AN INTERNATIONAL APPLICATION
==

To reach worldwide audiences with your products, you need to create applications that can be marketed in more than one country and that can be modified for new markets.

An international application must have the following characteristics:

- Country and language independence

- Easy localization

All applications, regardless of the language used in the interface, should be able to handle data from different countries and in different languages. For example, a database developed primarily for the English-speaking market should be able to handle French and German input. The application also should handle the appropriate currency symbols and date and time formats. Furthermore, it should be able to execute complex operations, such as sorting, using the language selected by the user.

Ease of localization is the second goal to strive for when writing international applications. Localization can be defined as the process of adapting an application for a market other than the one for which it was originally designed. This adaptation involves translation of the product, addition of new features when required, and modification of the product to meet local needs. Applications should be developed so that localization is a fast and painless task.

This document explains how to internationalize your Windows application.

ACHIEVING COUNTRY AND LANGUAGE INDEPENDENCE

Windows 3.00 provides resources to help your applications achieve country and language independence. These resources consist of international information stored in the WIN.INI file and in language-sensitive Windows functions. By using these resources and following the guidelines described in this section, your application will produce the correct international behavior.

INTERNATIONAL INFORMATION IN WIN.INI

The [Intl] section of the WIN.INI file contains the current Windows country settings. These settings can be modified by the user through the Control Panel, or by the application through the WriteProfileString() function. Applications have access to the current country settings through the GetProfileInt() and GetProfileString() functions. Applications should read the required country settings at start-up, and should monitor the WM_WININICHANGE message to update its country settings accordingly, in case they are changed.

The following is a list of the country settings stored in WIN.INI:

Setting	Description
iCountry	Country code. This value is based on the telephone country code. The only exception is Canada, for which a 2 is used instead of 1 (1 is used by the United States). Use this setting if your application has to control a country-dependent feature not supported by Windows 3.00.
sCountry	String defining the selected country name.
sLanguage	The national language selected by the user. Changing the language using the Control Panel's International dialog box will change the installed language-dependent module. The language values are as follows:

Value	Language

dan	Danish
dut	Dutch
eng	International English
fcf	French Canadian
fin	Finnish
frn	French
ger	German
ice	Icelandic
itn	Italian
nor	Norwegian
por	Portuguese
spa	Spanish
swe	Swedish
usa	U.S. English

sList — List separator. This character is used to separate elements in a list. The list separator must be different from the decimal separator to avoid conflicts with lists of numbers.

iMeasure — Measurement system selected by the user, where 0 = metric, 1 = English. Use this setting to control measurement-dependent features of your application.

iTime — Time format. This setting defines the time format: 12 hours or 24 hours, where 0 = 12-hour clock, 1 = 24-hour clock.

sTime — Time separator. This character is displayed between hours and minutes, and between minutes and seconds.

s1159 — In some countries, the time is displayed followed by a trailing string (AM, for example). This setting contains the trailing string used for times between 00:00 and 11:59.

s2359 — Trailing string (PM, for example) for times between 12:00 and 23:59, when in 12-hour clock format, or trailing string (GMT, for example) for any time in 24-hour clock format.

iTLZero — When displaying time, this value specifies whether or not the hours should have a leading zero. The convention is 0 = no leading zero (9:15, for example), 1 = leading zero (09:15, for example).

iDate — This is the Windows 2.x style for defining the date format. It has been kept for compatibility reasons. We recommend using sShortDate instead. The values for this setting are:

0 = Month-Day-Year
1 = Day-Month-Year
2 = Year-Month-Day

sDate — Date separator. Kept for compatibility with Windows 2.x. Try using sShortDate instead.

sShortDate — This is a new Windows 3.00 format. This string defines a "date picture" of the short date format. More information about the short date format can be stored using this method. sShortDate accepts only the values M, MM, d, dd, yy and yyyy. See sLongDate for information about these values and pictures.

sLongDate — This setting is like sShortDate, but it also can contain strings mixed with days of the week, dates, months, and years. The definition of this date picture is as follows:

Value	Item	Format
M	Month	1-12
MM	Month	01-12
MMM	Month	Jan-Dec
MMMM	Month	January-December
d	Day	1-31
dd	Day	01-31
ddd	Day	Mon-Sun
dddd	Day	Monday-Sunday
yy	Year	00-99
yyyy	Year	1900-2040

Examples:

Date Picture	Meaning
d MMMM, yyyy	9 January, 1989
dddd, MMMM d, yyyy	Friday, February 7, 1989
M/d/yy	3/18/89
dd-MM-yyyy	18-03-1989
d 'of' MMMM, yyyy	9 of January, 1989

sCurrency — This string defines the currency symbol of a given country. Be very careful with this setting. Do not make global replacements of currency amounts in your application if the currency symbol is changed through the Control Panel. Once the user has entered an amount using certain currency, that currency should stay the same. Also, be careful with this setting when sharing files among users or applications.

iCurrency — This setting defines the currency format. The convention is:

0 = Currency symbol prefix, no separation ($1, for example)
1 = Currency symbol suffix, no separation (1$)
2 = Currency symbol prefix, one character separation ($ 1)
3 = Currency symbol suffix, one character separation (1 $)

iCurrDigits This value defines the number of digits used for the fractional part of a currency amount.

iNegCurr This value defines the negative currency format. The definition follows the convention:

0 = ($1)
1 = -$1
2 = $-1
3 = $1-
4 = (1$)
5 = -1$
6 = 1-$
7 = 1$-

In these examples, the dollar symbol represents any currency symbol defined by sCurrency.

sThousand This is the symbol used to separate thousands in numbers with more than three digits.

sDecimal Character used to separate the integer part from the fractional part of a number.

iDigits Value defining the number of decimal digits that should be used in a number.

iLzero This setting defines whether a decimal value less than 1.0 (and greater than -1.0) should contain a leading zero.

0 = No leading zero (for example, .7)
1 = Leading zero (0.7)

LANGUAGE-SENSITIVE WINDOWS FUNCTIONS

Windows 3.00 introduces the concept of national language. Language, in conjunction with country, allows Windows to describe more precisely the characteristics of a given geographical location. The following is a list Windows functions that behave differently, depending on the language selected:

AnsiLower()
AnsiLowerBuff()
AnsiUpper()
AnsiUpperBuff()
IsCharAlpha()
IsCharAlphaNumeric()
IsCharLower()
IsCharUpper()
lstrcmp()
lstrcmpi()

Comparing and Sorting Strings

The Windows 3.00 functions lstrcmp and lstrcmpi allow applications to compare and/or sort strings based on the natural language selected by the user. These functions take into account different alphabetical orderings, diacritical marks, and special cases that require character compression or expansion.

It is very important to notice that these functions do not act the same way as do the C functions strcmp and strcmpi. The comparison done by lstrcmp and lstrcmpi is based on a primary value and a secondary value (see the following table). Each character has a primary and a secondary value. For example, in the following matrix, the letter "d" has a primary value of 4 and a secondary value of 2.

Primary Values	Secondary Values													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	A	A2	A3	A4	A5	A6	A7	a	a2	a3	a4	a5	a6	a7
2	B	b												
3	C	C2	c	c2										
4	D	d												
5	E	E2	E3	E4	E5	e	e2	e3	e4	e5				
6	F	f												

****
NOTE: This table uses these character values because some accented characters cannot easily be represented for electronic transmission. The printed application note contains the actual accented characters and may be easier to read and comprehend. Capital letters precede the lowercase letters. The following is a list of the accent codes:

A2 - A with a grave accent
A3 - A with an acute accent
A4 - A with a circumflex
A5 - A with a tilde
A6 - A with an umlaut
A7 - A with a circle
C2 - C with a cedilla
E2 - E with a grave accent
E3 - E with an acute accent
E4 - E with a circumflex
E5 - E with an umlaut
****

Examples of Primary and Secondary Sorting Values

When performing the comparison of two strings, the primary value takes precedence over the secondary value. That is, the secondary value is ignored unless a comparison based on the primary value shows the strings as equivalent.

The following examples show the effect of primary and secondary values on string comparisons:

Comparison	Reason
A = A	Primary values equal
A < a	Primary values equal, secondary values unequal (A < a)
Ab < ab	Primary values equal, secondary values unequal (A < a)
ab < Ac	Primary values unequal (b < c)

Note, however, that lstrcmpi ignores the effect of case in determining secondary value. That is, when lstrcmpi is called to compare "AB" and "ab", the two strings will be equivalent. However, lstrcmpi does not ignore diacritical marks, so "Ab" precedes "(a6)b", regardless of whether the comparison is performed by lstrcmp or lstrcmpi. ("a6" is an "a" with an umlaut.)

When comparing strings of different lengths, length takes precedence over secondary values. That is, the shorter string will always precede the longer string as long as the primary values in the shorter string equal the primary values of the equivalent characters in the longer string. For example, "ab" precedes "ABC", but "ABC" precedes "AD".

Depending on the language module installed, some characters will be treated differently. For example, if the German language module is installed, the beta character expands to "ss". If the Spanish language module is installed, the characters "ch" will be treated as a single character that sorts between "c" and "d".

Case Conversions

The case conversion functions AnsiLower(), AnsiLowerBuff(), AnsiUpper() and AnsiUpperBuff() depend on the language module installed. Different languages treat case conversions differently. Do not use the C case-conversion functions; they do not take into consideration characters with values more than 128.

Character Classification Functions

The functions IsCharAlpha(), IsCharAlphaNumeric(), IsCharLower(), and IsCharUpper() are also language dependent. Use these functions to attain language independence.

Handling Character Sets: ANSI Versus OEM

One of the main problems developers face when writing international Windows applications is handling characters sets. It is very important to understand ANSI and OEM.

ANSI is the character set used internally by Windows and its

applications. Windows does not recognize any character set other than ANSI.

OEM is defined by Windows as the character set used by DOS. The term "OEM" does not refer to a specific character set; instead, it refers to any of the different character sets (code pages) that can be installed and used by DOS.

Because Windows runs on top of DOS, there must be a layer between Windows and DOS that performs translations between ANSI and OEM. When Windows is first installed, the Windows Setup program looks at the DOS-installed character set, and then installs the correct ANSI-OEM translation tables and Windows OEM fonts.

Windows applications should use the Windows functions AnsiToOem() and OemToAnsi() when transferring information to and from DOS. Also, applications should use the correct character set when creating filenames. For more information about handling filenames, see the following section.

There is no one-to-one mapping between ANSI and OEM. Applying AnsiToOem() and then OemToAnsi() to a given string will not always result in the original string.

Keep in mind that both ANSI and OEM are 8-bit character sets. Always use "unsigned char" instead of "signed char". Bugs that result from using "signed char" are very hard to track.

Handling Filenames

One of the problems dealing with the ANSI and OEM character sets is the handling of filenames. Different applications do file handling differently, depending on factors such as speed, size, and programming style. This section describes the most common methods.

The easiest way to deal with filenames in Windows is to use ANSI for all filenames, and use the functions _lcreat(), _lopen(), and OpenFile() to deal with DOS and the OEM character set.

Another way to deal with filenames is to use OpenFile() to obtain a fully qualified pathname, the szPathName field, from the OFSTRUCT data structure. Be very careful here. The szPathName field contains characters from the OEM character set. The szPathName field must first be converted to ANSI before it is used as a parameter for OpenFile(), other Windows functions, or in a dialog box.

The following example shows this conversion:

```
if (OpenFile("myfile.txt", &of, OF_EXISTS) == -1)
    {
        OemToAnsi(of.szPathName, szAnsiPath);
        OpenFile(szAnsiPath, &of, OF_CREATE);
    }
```

Note that the value of of.szPathName must be converted from OEM to

ANSI.

The third, and perhaps most complicated, method of handling files is to directly call DOS [using the DOS3Call() function or an INT 21H instruction]. You must ensure that your application always passes OEM characters to DOS.

Another problem occurs when applications try to create filenames in ANSI that have no equivalent characters in OEM. For example, the character E4 (E-circumflex) does not exist in code page 437 (437 is the standard U.S. extended ASCII character set). If the application tries to save the file (E4).TXT, Windows will convert (E4).TXT into E.TXT [by using the AnsiToOem() function], and then it will pass the file to DOS. The end result is a confused user that doesn't understand what happened to his/her file. You can solve this problem by using the ES_OEMCONVERT and CBS_OEMCONVERT control styles. These styles (the first for edit controls and the second for combo boxes) will read the user's input and convert the typed character to a valid character (one that exists in the OEM character set). This way, the user will see on the screen the real filename that will be stored at the DOS level.

Handling the Keyboard

The most important keyboard issue for international applications is the use of the VK_OEM keys as user input. The problem here is that the locations of the VK_OEM keys change, depending on the keyboard layout chosen by the user. The VkKeyScan() function is helpful in these cases.

VkKeyScan() is used to translate an ANSI character into a virtual-key code plus a shift state. This function also could be used when one application has to send text to another application by simulating keyboard input.

Some other useful functions are the following:

- ToAscii(). This function is the opposite of VkKeyScan(). It converts a virtual-key code plus a shift state to an ANSI character.

- GetKeyNameText(). This function retrieves a string that contains the name of a key (for example, the SHIFT key or the ENTER key). The string will be in the language related to the keyboard. For the French keyboard layout, the name of the keys will be in French.

- GetKbCodePage(). It is important to note that there is no real relationship between the keyboard and the code page installed. This function will return the code page (OEM character set) that was running at the DOS level when Windows was installed.

To enter characters that are not on your keyboard, use the ALT key and the numeric keypad. For ANSI characters, hold down the ALT key and then, on the numeric keypad, type 0 (zero) and the three-digit code of the character you want. For OEM characters, do the same thing without typing the 0 prefix.

Handling WIN.INI, SYSTEM.INI, SETUP.INF, and Private Initialization Files

The WIN.INI, SYSTEM.INI, and SETUP.INF files are ANSI files. Normally, applications do not touch SYSTEM.INI or SETUP.INF. For WIN.INI and private initialization files, applications should use the functions GetPrivateProfileInt(), GetPrivateProfileString(), GetProfileInt(), GetProfileString(), WritePrivateProfileString(), and WriteProfileString(). Make sure ANSI is always used with these functions.

The section names and setting names in WIN.INI and private initialization files should be independent of the language of the application. Normally, all of these names should be in English. For example, in WIN.INI, the section name [Desktop] and the setting name Wallpaper should always remain in English so that applications in different languages can access the same information.

ACHIEVING EASY LOCALIZATION

Creating applications that are easy to localize is not difficult if you follow a few basic rules.

ISOLATION OF LOCALIZABLE INFORMATION

The most important rule for localization is to never mix functional code with strings, messages, or any other information that has to be modified. In a normal Windows application, all the menus, strings, and messages should be placed in the resource script (.RC) file. All the dialog-box information should be placed in the dialog script (.DLG) file. If you do this, there will be no need to recompile the executable file for a new localized version of the product. Just use the resource compiler (RC). Hard-coded strings (strings mixed with functional code) are the worst enemy of localization.

Strings that are not meant to be modified (filenames, WIN.INI setting names, etc.) can be placed in the resource script file. In this case, the .RC file should contain comments documenting that the names are permanent. Better yet, mark what has to be translated (explaining limitations, if any) and what should not be modified. The better the documentation, the easier the localization.

Place in the .RC files and .DLG files anything that could be a localization item. It is better to have extra information in these files than to have too little.

In cases where an .RC or .DLG file cannot be used, place all the information in a file (such as an include file) that is separate from any functional code.

ALLOCATING EXTRA SPACE FOR STRINGS

Many languages are more verbose than English; therefore, they require more space to hold strings or to display dialog boxes. There are cases, as with menus, where the space allocation is done dynamically. However, in most cases, the application must provide the space. The following table shows how much additional space should be allocated for strings of various lengths:

Length of English Text (In Characters)	Space Allocation (In Addition to Text)
1-10	200 percent
11-20	100 percent
21-30	80 percent
31-50	60 percent
51-70	40 percent
70+	30 percent

Avoid creating dense menus where most of the available space (a line, for example) is already used up in the English version. Dialog boxes should be designed so that items can be moved freely, allowing the organization of the contents as the translation demands. Do not crowd status bars with information. Even abbreviations are often longer in different languages.

HANDLING FOREIGN SYNTAX AND GRAMMAR

Never make assumptions about syntax or grammar when dealing with foreign languages. The ordering of words can be different, and the number of words required is often greater than in English.

All messages should be self contained, not dynamically assembled. For messages that have variables added to them at run time, do not make any assumptions about the position of the variable in the message. The way to handle variables in messages is by using the Windows function wsprintf(). For example, you could place in the .RC file the string containing the variable, as follows:

```
CannotOpen, "The application could not open the file %s"
```

Use wsprintf() to incorporate the variable into the string, as follows:

```
LoadString(hInst, CannotOpen, lpFormat, MaxLen);
wsprintf(FinalString, lpFormat, FileName);
```

Avoid using a single word in more than one message. Words such as "None" can have different translations (different gender and number) depending on the context.

Do not create plurals of words by adding "s". Keep two strings, one for the singular and one for the plural.

Avoid parsing text to obtain information. Parsing normally assumes specific syntax.

Avoid using slang, abbreviations, and jargon, since they are difficult to translate.

When handling graphic objects such as bitmaps, cursors, and icons, try to avoid the use of embedded text. Text is difficult to modify when in graphical form. If you cannot avoid this, be careful about leaving enough space for translation, and try to create tools to simplify the modification.

Graphic objects are also language dependent. Always look for graphic objects that represent international concepts.

OTHER RULES

Do not hard code the position or size of any element on the screen. Remember that items will change position and size as they get translated. If you must define the size or position of certain object, place this definition in the .RC file.

Be careful when using the CreateWindow() function. This function contains two parameters: lpClassName and lpWindowName. The lpClassName parameter should be constant and independent from localization. On the other hand, lpWindowName is the string that will appear in the caption bar and therefore should be localized. The string used for lpWindowName should be taken from the resources.

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrLocUnicode

INF: Extended Characters Different Under Windows
Article ID: Q83461

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Applications in the Windows environment must typically deal with two different character sets: the ANSI (American National Standards Institute) character set and the OEM (original equipment manufacturer) character set. Conversely, applications in the MS-DOS environment must deal only with the OEM character set. This article describes how Windows deals with the ANSI and OEM character sets.

1. When ALT+xxx is used to enter a character from the OEM character set into an application in the Windows environment that uses the ANSI character set, Windows displays the character in the ANSI character set that most closely matches the entered character.

2. When a character from the OEM character set is entered into a file using a text editor under MS-DOS and the file is displayed under Windows, the character from the ANSI character set that has the same character code number as the OEM character is displayed.

More Information:

OEM and ANSI Character Sets

MS-DOS uses the OEM character set. This character set varies between computers and depends on the code page ROM (read-only memory) installed by the computer manufacturer. For example, personal computers manufactured for use in the United States use a character set called code page 437, while computers manufactured for use in Portugal use code page 860. MS-DOS uses the OEM character set in applications and to create files and filenames.

For the most part, Windows uses fonts organized according to the ANSI character set (called ANSI-set fonts, in this article). Windows also supports fonts that use the same OEM character set that MS-DOS uses (called OEM-set fonts, in this article).

Character positions 32 through 127 are identical in the ANSI and OEM character sets for most code pages (including code pages 437, 850, 852, 860, 861, and 865). The remaining characters of the OEM character set (character positions 0 through 31 and 128 through 255) either do not appear in the ANSI character set, or exist at a different position in the ANSI character set. Therefore, some characters in the OEM character set cannot be displayed in Windows using an ANSI-set font. If an application must display such characters under Windows, an OEM-set font is required.

Typing ANSI and OEM Characters in Windows

In the Windows environment, a user can enter any character in the character set by holding down the ALT key and typing 0xxx, where "xxx" is the decimal number of the desired character position in the font. For example, with an ANSI-set font in use, ALT+0123 will display the 123rd character in the ANSI character set. Similarly, with an OEM-set font in use, ALT+0123 will display the 123rd character in the OEM character set.

In the MS-DOS environment, a user can enter any character in the OEM character set by holding down the ALT key and typing xxx (no leading zero), where "xxx" is the decimal number of the desired character position in the font.

If a user enters an MS-DOS OEM character set code (ALT+xxx) in an application for Windows that uses an ANSI-set font, Windows converts the OEM-set character to the character that most closely matches in the ANSI set. This conversion is governed by a mapping table that is installed with Windows. Because some OEM-set characters with positions greater than 127 do not exist in the ANSI character set, the result of the conversion in Windows may differ from the character in the OEM set. The OemToAnsi function uses the same mapping table to perform its character conversions.

For example, while OEM character-set code page 437 contains a square-root symbol at position 251, the ANSI character set does not contain this character. Consequently, when the user types an ALT+251 in an edit control that uses the ANSI character set, an underscore character appears because Windows defines the character mapping in this manner. As another example, the C-cedilla character exists in both the ANSI character set and in the OEM character-set code page 437. Therefore, typing ALT+128 in an edit control creates the desired C-cedilla character. Note that while the character exists in both character sets, its position is different in each set (128 in the OEM character set and 199 in ANSI). The alternative method to request a C-cedilla is to type ALT+0199, which specifies the character's position in the ANSI character set.

An edit control that uses the ES_OEMCONVERT style and a combo box that uses the CBS_OEMCONVERT style have a different behavior from that described above. These two styles cause their text contents to be converted from lowercase letters to uppercase letters, then from the ANSI set to the OEM set and then back to the ANSI set for display. This behavior is important for an edit control in which the user specifies a filename. If the user enters characters that do not exist in the underlying OEM character set, the name of the file will differ from the name specified by the user, which would be confusing. Because the characters are mapped into characters that exist in the OEM character set, the filename specified always matches the filename actually used. The contents are converted to uppercase characters because it is customary in some languages to eliminate diacritical marks when a character is in uppercase, and the OEM character set does not contain uppercase characters with these diacritical marks.

Displaying a String Containing OEM-Set Characters in an Application that Uses the ANSI Character Set

Text editors running under MS-DOS use the OEM character set for display and in the files they create. When a Windows-based text editor loads a file that uses the OEM character set, the editor interprets the characters according to the ANSI character set. Character positions 32 through 127 are not affected under most code pages because both the ANSI and OEM character sets have identical characters. However, character positions greater than 127 may be displayed differently than in the MS-DOS-based text editor because the character positions represent different characters in the ANSI character set.

The solution to this difficulty is to use a Windows-based text editor that uses the ANSI character set when the text contains characters in both the OEM and ANSI character sets. A Windows-based editor accepts ANSI-set characters directly and converts OEM-set characters to the closest matching ANSI-set characters. The resulting text contains only ANSI-set characters, which can be displayed by any application running under the Windows environment. If an application must display OEM-set characters that are not in the ANSI character set, it must use an OEM-set font.

Consider the following example: An MS-DOS-based text editor is used to edit a application's resource file on a system with OEM character-set code page 437 installed. The user types ALT+129 as part of the static text to label a button in a dialog box. However, when the dialog box is displayed, the text is not as expected but contains a black rectangle where the u-umlaut character belongs. The black rectangle is used to signify character positions that are not defined in the ANSI character set.

To workaround to this problem is to edit the resource file with a Windows-based text editor that uses the ANSI character set. Typing ALT+129 will create a u-umlaut as desired because the editor will convert the OEM-set character to the closest matching ANSI-set character. In this case OEM-set character position 129 maps to ANSI-set character position 252. The alternative method to specify u-umlaut in the Windows-based editor is to type ALT+0252, using its ANSI character set character position directly.

As another example, an application requires the square-root symbol, which does not exist in the ANSI character set, as part of a button label. Assuming the code page 437 is installed, and that the resource file is edited under Windows, enter ALT+0251 in the button label because the square-root symbol is the 251st character of the OEM character set. When the application is run, send a WM_SETFONT message to the control, specifying an OEM-set font. An OEM-set font is always available from the GetStockObject function through its OEM_FIXED_FONT index.

For more information on code pages and character sets under Windows, query on the following words in the Microsoft Knowledge Base:

prod(winsdk) and code and pages and character and sets

For a reference to a Windows Developer's Note regarding this subject,
query on the following word in the Microsoft Knowledge Base:

INTLAPPS

Additional reference words: 3.00 3.10 folding
KBCategory:
KBSubcategory: UsrLocAnsi/oem

PRB: IsCharAlpha Return Value Different Between Versions

Article ID: Q84843

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

Under Windows version 3.1, the IsCharAlpha function returns TRUE for the character values 8Ah, 8Ch, 9Ah, 9Ch, 9Fh, and DFh. Under Windows version 3.0, the function returns FALSE for these character values.

CAUSE

These characters represent alphabetic characters that were added to the Windows character set in Windows 3.1.

RESOLUTION

Applications that use the IsCharAlpha function should behave properly with the newly-defined characters. No changes should be required.

More Information:

Appendix C.1, page 596, of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 3: Messages, Structures, and Macros" lists the Windows character set.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrLocStringsor

PRB: Multikey Help Code Incorrect in Windows Tools Manual

Article ID: Q69896

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

The sample code to access an alternate keyword table in a Windows Help file from an application, provided on page 18-21 in the "Microsoft Windows Software Development Kit Tools" manual for version 3.0 is incorrect.

STATUS/RESOLUTION

Microsoft has confirmed that this documentation error occurs on page 18-21 in the "Microsoft Windows Software Development Kit Tools" manual for version 3.0. This error has been corrected on Page 42 of the "Microsoft Windows Software Development Kit Tools" manual for version 3.1. A different example is included on Page 42.

The following is the correct code to access the 'B' keyword table in a Windows Help file:

```
HANDLE hmk;
MULTIKEYHELP far *pmk;
char szKeyword[] = "B-keyword";

case MULTIKEY:
   hmk = GlobalAlloc(GHND, (sizeof(MULTIKEYHELP)+lstrlen(szKeyword)));
   if (hmk == NULL)
       break;
   pmk = (MULTIKEYHELP FAR *)GlobalLock(hmk);
   pmk -> mkSize = sizeof(MULTIKEYHELP)+lstrlen(szKeyword);
   pmk -> mkKeylist = 'B';
   lstrcpy(pmk -> szKeyphrase, szKeyword);

   WinHelp(hWnd, szHelpFileName, HELP_MULTIKEY, (DWORD)(LPSTR)pmk);

   GlobalUnlock(hmk);
   GlobalFree(hmk);

   break;
```

Additional reference words: 3.00 docerr MICS3 T18
KBCategory:
KBSubcategory: TlsHlpMisc

INF: UNINPUT - Sample Application

Article ID: Q103296

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

UNINPUT: Demonstrates Primitive Drag-and-Drop Unicode Input Method

UNINPUT provides a primitive, mouse-based, drag-and-drop input method. It allows the user to grab any character covered by a Unicode font, and drag the character to a second window. If the user drops the character on the second window, that window is sent a WM_CHAR message with the appropriate key code.

The status bar at the bottom of the window displays three fields of information:

- The title of the last window to receive a WM_CHAR (left)
- The type of the last window to receive a WM_CHAR--either Unicode or ANSI (center)
- A list of the most recently dropped characters--similar to a history buffer (right)

Please see the online help file for more information.

The Microsoft(R) Windows NT(TM) version 3.1 operating system was not available at the time this sample was acquired for the Microsoft Developer Network CD. For this reason, the CD includes only the source files for the sample, not the executable file or any DLLs. If you have Windows NT version 3.1 installed on your system, you may copy the source files to your hard drive and invoke the Windows NT make file to build the sample.

KEYWORDS: CD4

For Microsoft OnLine customers, UNINPUT is available in the Software/Data Library as 4X80.ZIP and can be found by searching on the word UNINPUT, the Q number of this article, or S14255. For CompuServe Customers, UNINPUT can be downloaded from the file 4-80.ZIP in library section number 4 of the MSDNLIB forum. UNINPUT was archived using the PKware file-compression utility.

Additional reference words: 3.10
KBCategory:
KBSubcategory:

INF: DBCS Support in Windows Versions 3.0 And 3.1

Article ID: Q75255

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In a version of Windows that is enabled for double-byte character sets (DBCS), all functions that take strings as arguments will work with strings of double-byte characters. In addition, any version of Windows that supports DBCS will support the IsDBCSLeadByte function. This function determines whether or not a byte is the leading byte of a 2-byte character. The Windows functions that use a string length parameter (like lstrlen and TextOut) will use a count of bytes, not of characters. The functions IsDBCSLeadByte, AnsiPrev, and AnsiNext should be used to access characters within a string.

Note that DBCS-enabled Windows will typically have special methods to enter double-byte or composite characters. Composite character creation involves pressing several keys in a specified sequence to form new characters. The entry system is integrated into Windows by the original equipment manufacturer (hardware OEM). These characters are not typically directly entered into an edit control. Composite characters may be entered at the bottom of dialog box (as in OS/2). In many systems, these characters are composed at the bottom of the screen (in the composition window).

Copies of Windows version 3.0 developed for the United States and Europe do not support DBCS. Copies of Windows version 3.1 developed for the United States and Europe, and the KANJI adaptation of Windows version 3.0 and 3.1 do support DBCS. The KANJI adaptation of Windows is distributed through hardware OEMs. For more information on KANJI Windows, search this knowledge base on the following words:

   prod(winsdk) and KANJI

The March, 1990, "Microsoft Systems Journal" article "Using the OS/2 National Language Support Services to Write International Programs" has more information on both DBCS and KANJI. It can be found in the Software/Data Library by searching on OS2NLS or S12547. OS2NLS was archived using the PKware file-compression utility.

Additional reference words: 3.00 3.10 3.x softlib
KBCategory:
KBSubcategory: UsrLocDbcs

PRB: Accented Characters in Filename Controls Lose Accents

Article ID: Q90854

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.0 and 3.1

Summary:

SYMPTOMS

In an application for the Microsoft Windows graphical environment that uses a combo box with the CBS_OEMCONVERT style or an edit control with the ES_OEMCONVERT style, the user enters a character with an accent mark, and the accent is lost.

CAUSE

The OEMCONVERT control styles convert all input as follows:

- Convert lowercase letters to uppercase letters with the AnsiUpper function.

- Convert all characters from the ANSI character set to the installed OEM character set with the AnsiToOem function.

- Convert the appropriate characters back to lowercase letters with the AnsiLower function.

Because the OEM character set installed in most computers (code page 437) does not include representation for the capital letters with accent marks, the accent marks are lost in this conversion process. This occurs in all versions of Windows, including those designed for use outside the United States.

RESOLUTION

Unicode specifies a unified character set that can represent every active language and many dead languages. Windows NT incorporates support for Unicode.

More Information:

For further details on the cause of this problem, see the article by Dr. William S. Hall "Adapt Your Program for Worldwide Use with Windows Internationalization Support," starting on page 29 of the Nov-Dec 1991 Microsoft Systems Journal.

Additional reference words: 2.00 2.03 2.10 2.x 3.00 3.10
KBCategory:
KBSubcategory: UsrCtlCombo

INF: Tips for Writing DBCS-Compatible Applications
Article ID: Q75439

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

String operations in systems that use a double-byte character set (DBCS) are slightly different from a single-byte character system. This article provides guidelines to reduce the work necessary to port an application written for a single-byte system to a DBCS system.

More Information:

In a double-byte character set, some characters require two bytes, while some require only one byte. The language driver can distinguish between these two types of characters by designating some characters as "lead bytes." A lead byte will be followed by another byte (a "tail byte") to create a double-byte character (DBC). The set of lead bytes is different for each language. Lead bytes are always guaranteed to be extended characters; no 7-bit ASCII characters can be lead bytes. The tail byte may be any byte except a NULL byte. The end of a string is always defined as the first NULL byte in the string. Lead bytes are legal tail bytes; the only way to tell if a byte is acting as a lead byte is from the context.

The Windows Software Development Kit (SDK) version 3.0 includes two functions for moving through strings that may contain DBCs: AnsiNext() and AnsiPrev(). The AnsiPrev() function is a time expensive call because it must run through the string from the beginning to determine where the previous character begins. It is best to search for characters from the beginning rather than the end of a string.

The Windows SDK version 3.1 includes the IsDBCSLeadByte() function, which returns TRUE if and only if the byte CAN BE a lead byte. Because this function takes a char parameter, it cannot report if the byte IS a lead byte (to do so would require context).

To make non-DBCS code run as quickly as possible, a source file may use "#ifdef DBCS" around code that is only for DBCS, and compile two versions of the object (OBJ) file. For example:

```
#ifdef DBCS
  for (pszTemp = szString; *pszTemp; pszTemp = AnsiNext(pszTemp))
#else
  for (pszTemp = szString; *pszTemp; ++pszTemp)
#endif
   ...
```

To make the code easier to read, an application could define macros for the AnsiNext() and AnsiPrev() functions if DBCS is not defined:

```
#ifndef DBCS
#define AnsiNext(x) ((x)+1)
#define AnsiPrev(y, x) ((x)-1)

#ifdef WIN31
#define IsDBCSLeadByte(x) (FALSE)
#endif

#endif
```

With these definitions in place, all of the code can be written for DBCS. Note that the AnsiNext() function will not go past the end of a string and the AnsiPrev() function will not go past the beginning of a string, while the macros will. In addition, because the "y" parameter in the AnsiPrev() macro is ignored, some code will give different results when compiled with and without DBCS defined. The following code is an example of this phenomenon:

```
   pszEnd = AnsiPrev(++pszStart, pszEnd);
```

The following code demonstrates how to find the offset of the filename in a full path name:

```
LPSTR GetFilePtr(LPSTR lpszFullPath)
{
    LPSTR lpszFileName;

    for (lpszFileName = lpszFullPath; *lpszFullPath;
               lpszFullPath = AnsiNext(lpszFullPath))
        if (*lpszFullPath == ':' || *lpszFullPath == '\\')
            lpszFileName = lpszFullPath + 1;

    return lpszFileName;
}
```

Note that ':' and '\\' are guaranteed not to be lead bytes. The search started from the beginning of the string rather than the end to avoid using the AnsiPrev() function.

The following code demonstrates a string copy into a limited size buffer. Note that it ensures that the string does not end with a lead byte.

```
int StrCpyN(LPSTR lpszDst, LPSTR lpszSrc, unsigned int wLen)
{
    LPSTR lpEnd;
    char cTemp;

    // account for the terminating NULL
    --wLen;

    for (lpEnd = lpszSrc; *lpEnd && (lpEnd - lpszSrc) < wLen;
               lpEnd = AnsiNext(lpEnd))
        ;  // scan to the end of string, or wLen bytes

    // The following can happen only if lpszSrc[wLen-1] is a lead
```

```
    // byte, in which case do not include the previous DBC in the copy.
    if (lpEnd - lpszSrc > wLen)
        lpEnd -= 2;

    // Terminate the source string and call lstrcpy.
    cTemp = *lpEnd;
    *lpEnd = '\0';
    lstrcpy(lpszDst, lpszSrc);
    *lpEnd = cTemp;
}
```

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrLocDbcs

INF: lstrcmpi, Accented Characters, and Sort Order in Windows

Article ID: Q100366

Summary:

This article provides supplementary information to Section 18.2.2.1 "Comparing and Sorting Strings" in the Windows version 3.1 Software Development Kit "Programmer's Reference, Volume 1: Overview" manual and the Windows Help International Overview section. Specifically, this article provides information about the sort order used by the Windows lstrcmp and lstrcmpi functions, the location of accented characters in the sort, and how primary and secondary values, or weights, are important when sorting a string using these functions. This includes the different behavior of these functions when a language driver is installed compared to when Windows's internal sort routine (English/American) is used.

List boxes that include the LBS_SORT style use lstrcmpi internally to perform the sort. Consequently, the information in this article applies to these list boxes also.

ALPHSORT is a sample that shows the results of these sort routines, and contains a dialog box with two list boxes. One list box contains the characters 32-255 and the option to display the characters in ANSI or sorted order; the other list box includes the LBS_SORT style and contains various strings. Changing the sort routine in use through the International Control Panel application illustrates the effects of the different language drivers and Windows's internal sort routine.

ALPHSORT can be found in the Software/Data Library by searching on the word ALPHSORT, the Q number of this article, or S14194. ALPHSORT was archived using the PKware file-compression utility.

More Information:

The sort order used by lstrcmp and lstrcmpi is:

1. Nonalphanumeric characters (in ASCII/ANSI order)
2. Numeric characters
3. Alphabetic characters

For performance reasons, the internal sort routine treats accented characters as nonalphanumeric. Therefore, when the internal routine is used, accented characters appear towards the beginning of the sort between punctuation and numbers. In contrast, when a language driver is used, accented characters appear near their unaccented equivalents because the language drivers sort accented characters as alphabetic characters.

The following illustrate the differences in character order ("..." signifies omitted characters):

ANSI Order:

```
!"#...0...9:;<...ABC...XYZ[\]...abc...xyz{|}...accented characters
```

Internal (English/American) Sort Routine Order:

```
!"#...:;<...[\]...{|}... accented characters 0...9AaBbCc...XxYyZz
```

Language Driver Order:

```
...!"#...:;<...[\]...{|}...0...9A accented characters aBbCc...XxYyZz...
```

Note that the accented characters are intermixed with their alphabetic counterparts here.

Primary and secondary weights of characters also affect the sort order when a language module is installed. In this case, sorting is done by primary weight for the entire length of the string, then by length, and lastly by secondary weight if the primary weights of all the characters and the lengths of the strings are equal. The secondary (diacritic) weights are important only when there is a tie in the entire string. The internal (English/American) sort routine does not sort extended characters; as mentioned above, they are sorted as punctuation rather than alphabetic characters. Therefore, the internal routines produce completely different results than the language routines in some cases.

Character weights are important with case-sensitive sorting also. For example, using lstrcmp will produce: A < a < B < b; it will also produce: Aaa < aaa < Aab. These examples use proper dictionary sort order, but the second example is not necessarily obvious because if A < a, then it seems Aab < aaa should also be true. In that case, it is said that A and a "collide" (that is, their primary weights are the same) and a delayed comparison must be performed if the remainders of the strings are equal. The strings continue to be compared character by character. Because a < b, then aaa < Aab and the comparison is complete. If the strings were equal all the way through (such as Aaa and aaa) then A and a would collide once again. The rest of the strings would be equal, and then the secondary weights of A and a would be checked to determine that Aaa < aaa.

EXAMPLES WHEN A LANGUAGE DRIVER IS INSTALLED

Note: Special notation is used below to represent accented characters due to limitations in the distribution media for this article. For example <u umlaut> is used to represent the letter "u", which has an umlaut over it. Likewise, <a tilde> represents the letter "a" accented with a tilde.

Nonaccented Characters

The sort works on a character-by-character comparison, checking primary weights in a string. As soon as the primary (alphabetic) weights show one string greater than the other, the comparison stops. Therefore, sorted lists resemble the following:

```
a
aa
```

abba
ac
add
b
ba
baa

Accented Characters

The secondary (diacritic) weights are important only when there is a tie in the string. For example, in the following sorted list, the characters in the first two strings have identical primary sort weights ("s", "a", "m", "e") and the strings are the same length. Because of the tie in primary weights and string lengths, the secondary weights are then compared. The first difference in secondary weight ("a" versus <a tilde>) breaks the tie. Secondary weights are also a factor when comparing strings 4 and 5 below:

1. same
2. s<a tilde>me
3. sandy
4. schon
5. sch<o umlaut>n
6. school

It is important to apply the primary weights to the whole string first, and only use the secondary weights in a tie. The primary weights must also carry more importance than the secondary weights as well. Otherwise, an incorrect sort would result (such as schon less than school less than sch<o umlaut>n). This type of weighting creates a sort that makes a distinction between "unique," hard-coded letters of a language and mere variants of a letter (which are only distinguished by diacritics).

Here is a sorted list using the English (International) driver:

a
<a grave>
apple
<a grave>pple
apples
<a grave>pples
lu
l<u umlaut>
l<u umlaut>b
lum

Here is the same list, using the internal [English (American)] routine:

<a grave>
<a grave>pple
<a grave>pples
a
apple
apples

    l<u umlaut>
    l<u umlaut>b
    lu
    lum

For more information on the international sort, see the Canadian Standards Association Z243.4.1-1990; "Canadian Alphabetic Ordering Standard for Character Sets of CSA Z243.4 Standard," Alain LaBont<e acute>:

DIN Standard 5007, "Orden von Schriftzeichenfolgen: ABC-Regeln," April 1991

IBM Document GG24-3516, "Keys to Sort and Search for Culturally Expected Results," Denis Garneau

"R<e acute>gles du classement alphab<e acute>tique en langue fran<c cedilla>aise...", Alain LaBont<e acute>

Additional reference words: 3.00 3.10 listboxes
KBCategory:
KBSubcategory: UsrLocStringsor

INF: Detailed Description of lstrcmp

Article ID: Q96748

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

Executing the following function in the English language returns -1:

```
lstrcmp("e","Z");
```

RESOLUTION:

lstrcmp performs a comparison in the same manner as does a real dictionary. Characters are compared in a case-insensitive manner first, then specific details are compared (for example, lowercase versus uppercase).

lstrcmp performs its comparisons by first setting up a primary and secondary weight. The primary weight is the value of the character after converting to uppercase. The secondary weight describes whether the character was originally lowercase or uppercase.

Using the standard ASCII table as a guide (0...9ABC...XYZabc...xyz), the following are examples of lstrcmp:

```
"A" < "a" < "B" < "b"
"Aaa" < "aaa" < "Aab"
```

In the first example, "A" is less than "a" because:

1. Both characters are converted to uppercase.
2. The primary weights are compared and found equal.
3. Because the primary weights are equal, the secondary weights are compared. It is then determined that "A" is less than "a" because "a" is higher in the ASCII chart.

Also in the first example, "a" is less than "B" because:

1. Both characters are converted to uppercase.
2. The primary weights are compared and it is determined that "A" is less than "B" on the ASCII chart. No secondary weight comparison is performed.

In the second example, "Aaa" is less than "aaa" following the same logic shown in the comparison of "A" and "a".

When two characters are compared and they are equal, this is called a "collision," and "delayed comparison" will be performed using the secondary weight.

More Examples:

```
e,a  --> E,A --> E>A = 1
e,e  --> E,E --> E=E = 0  Because they are equal, the case is
                          checked (secondary weight).
e,z  --> E,Z --> E<Z = -1

e,A  --> E,A --> E>A = 1
e,E  --> E,E --> E=E = 1  Because they are equal, the case is
                          checked (secondary weight).
e,F  --> E,F --> E<F = -1
```

More Information:

If you need to sort by ASCII values using case sensitivity, use the C run-time function strcmp or _fstrcmp.

Also, Far East Windows applications adopt this dictionary sorting technique in English and local languages.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrLocStringSor

INF: DDEEXEC.RTF - Technical Article

Article ID: Q97408

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

DDE Execute Strings

Herman Rodent
Microsoft Developer Network Technology Group

Created: November 16, 1992

Abstract

The dynamic data exchange (DDE) protocol includes a feature designed to allow a DDE client application to send commands to a DDE server. The syntax of these execute requests and the way they should be handled by servers is not well documented and, consequently, varies slightly with different implementations.

The current DDE protocol has no provision for returning result information from an execute request, so a client application can only tell that the request failed, not why it failed.

This article proposes a consistent way to handle DDE execute command requests and a mechanism that allows result information to be returned. This article covers the following points:

- DDE execute command syntax

- Special characters in command arguments

- Unicode considerations for Microsoft Windows NT

- The returning of result information

For Microsoft OnLine customers, DDEEXEC.RTF is available in the Software/Data Library as 10X84.ZIP and can be found by searching on the word DDEEXEC.RTF, the Q number of this article, or S14142. For CompuServe Customers, DDEEXEC.RTF can be downloaded from the file 10-84.ZIP in library section number 10 of the MSDNLIB forum. DDEEXEC.RTF was archived using the PKware file-compression utility.

INF: Menu Operations When MDI Child Maximized

Article ID: Q71836

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Pop-up menus added to an MDI application's menu using InsertMenu() with MF_BYPOSITION will be inserted one position further left than expected if the active MDI child window is maximized. This behavior occurs because the system menu of the active MDI child is inserted into the first position of the MDI frame window's menu bar.

To avoid this problem, if the active child is maximized when a new pop-up is inserted by position, add 1 (one) to the position value that would otherwise have been used. To determine that the currently active child window is maximized, send a WM_MDIGETACTIVE message to the MDI client window. If the high word of the return value from this message contains 1, the active child window is maximized.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiMenus

INF: Modifying the System Menu of an MDI Child Window
Article ID: Q77930

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The system menu of an MDI child can be modified only after the WM_CREATE message for the MDI child is processed. If an application tries to modify the system menu of an MDI child during the processing of the child's WM_CREATE message, the system menu will not be changed.

More Information:

It is common to try to alter a window's system menu during the processing of its WM_CREATE message. This method has no effect on the system menu of an MDI child, however, because of the way a MDI child window is created:

1. The MDI client window calls the CreateWindowEx function to create the window.

2. Then, the MDI client window replaces the default system menu with the special MDI child system menu.

The WM_CREATE message is sent to the child window by CreateWindowEx. Changes to the MDI child's system menu on WM_CREATE have no effect because the MDI client assigns a special MDI system menu to the child after CreateWindowEx returns.

To work around this design decision, the child window procedure can post a custom message to itself during the processing of its WM_CREATE message. This custom message is processed after the MDI client has finished creating the MDI child. The code for this workaround might resemble the following:

```
#define WM_ADDMENUITEM (WM_USER+1)

LONG FAR PASCAL MDIChildWndProc(HWND hWnd,
                                WORD wMsg,
                                WORD wParam,
                                LONG lParam)
{
    switch (wMsg)
       {
    case WM_CREATE:
       PostMessage(hWnd, WM_ADDMENUITEM, 0, 0L);
       break;

    case WM_ADDMENUITEM:
       {
       HMENU        hMenu;
```

```
        hMenu = GetSystemMenu(hWnd, FALSE);
        InsertMenu(hMenu, -1, MF_BYPOSITION, IDM_NEW,
              (LPSTR)"New Item\0");
        DrawMenuBar(hWnd);
        }
        break;

    default:
        return DefMDIChildProc(hWnd, wMsg, wParam, lParam);
        }

    return 0;
}
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiMenus

INF: Terminating the Creation of an MDI Child Window
Article ID: Q80125

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In an application designed with the Microsoft Windows multiple document interface (MDI), there are situations in which it is desirable to destroy an MDI child window during the processing of its WM_CREATE message. For example, the design of the application may require terminating window creation if a memory allocation needed to hold data for an MDI child fails.

Unfortunately, returning -1 to end processing of the WM_CREATE message will not cleanly destroy the child window. Similarly, posting a WM_CLOSE message will cause the display to flash as the child is drawn, made active, and destroyed. This article describes a technique to avoid this unacceptable visual effect.

More Information:

To avoid the flash, the application can clear the redraw flag for the MDI client window. This prevents the MDI client window and its children from painting. The remainder of this article contains code to implement this technique.

Before the application processes the MDI child window's WM_CREATE message, it does not change its display to reflect the new child window. If the application determines that it must abort creating the MDI child window at this time, it should clear the MDI client window's redraw flag and post a WM_CLOSE message to the child window. When the application processes the WM_DESTROY message, turn the redraw flag back on. The following code demonstrates these steps:

```
case WM_CREATE:

    ...

    if (DestroyMePolitely)
        {
        SendMessage(hWndMDIClient, WM_SETREDRAW, FALSE, 0L);
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
        }
    break;

...

case WM_DESTROY:
    SendMessage(hWndMDIClient, WM_SETREDRAW, TRUE, 0L);
```

Additional reference words: 3.00 3.0

KBCategory:
KBSubcategory: UsrMdi

INF: Windows Does Not Support Nested MDI Client Windows

Article ID: Q74041

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The Microsoft Windows implementation of the multiple document interface (MDI) does not support nested MDI client windows. In other words, neither an MDI client window nor an MDI child window can have an MDI client window as a child.

More Information:

A Windows MDI client window is a member of the MDIClient window class, and the Windows MDI model assumes that the parent of an MDIClient window is a top-level frame window with a valid menu bar. This assumption is necessary to implement the basic functionality defined by the MDI interface, and it precludes the possibility of using nested MDIClient windows. However, an application can have multiple top-level windows, and each top-level window can have a separate MDIClient window as a child.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMdiCreatekid

PRB: Using Multiple Menus in an MDI Application

Article ID: Q67244

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Multiple document interface (MDI) applications with more than one class of MDI child windows commonly use a different menu to control windows of each class. This can be accomplished by selecting the appropriate class-specific menu each time an MDI child window becomes active.

This article describes two common programming errors that can cause MDI applications with multiple menus to work incorrectly. The two problems manifest themselves as follows:

Problem 1: Each time the MDI application terminates, the percentage of Free System Resources is lower. (The value for Free System Resources is displayed in the About Program Manager dialog box.)

Problem 2: Creating a new child window in a maximized state causes the menu bar to appear strangely and/or creates an Unrecoverable Application Error (UAE).

Charles Petzold's MDIDEMO sample application on pages 57-59 and 63 in the July 1990 issue of the "Microsoft Systems Journal" (Vol. 5, No. 4) exhibits the first problem. If MDIDEMO is altered to create child windows in a maximized state, it will exhibit the second problem.

The MDIDEMO application can be found in the Software/Data Library by searching on the keyword MDI, the Q number of this article, or S12629. MDIDEMO was archived using the PKware file-compression utility.

Below are descriptions of the solutions to these two problems.

==
Problem 1: Each time the MDI application terminates, the percentage of Free System Resources is lower.
==

Solution

This problem can occur if menus used by the MDI application are not destroyed before the application terminates.

Menus that are used by a program take up space in the shared segment that holds the USER heap. When the program terminates, Windows will

reclaim the memory used by the program's current menu. If there are other menus that are not currently in use, that memory must be reclaimed by calling DestroyMenu before the application terminates.

Example

MDIDEMO uses LoadMenu to load three menus: hMenuInit, hMenuHello, and hMenuRect. When MDIDEMO exits, the memory allocated for two of these menus is not reclaimed; about 1K is lost in the USER heap, and Free System Resources go down.

Adding this code to the FrameWndProc function in MDIDEMO will remove this problem:

```
   case WM_DESTROY:
     {
     HMENU hCurrentMenu = GetMenu(hwnd);

     if (hMenuInit != hCurrentMenu)
           DestroyMenu(hMenuInit);
     if (hMenuHello != hCurrentMenu)
           DestroyMenu(hMenuHello);
     if (hMenuRect != hCurrentMenu)
           DestroyMenu(hMenuRect);

     PostQuitMessage(0);
     return 0;
     }
```

Note that DestroyMenu must NOT be called on the menu that is currently in use, or the program will crash.

===
Problem 2: Creating a new child window in a maximized state causes the
           menu bar to appear strangely and/or creates an
           Unrecoverable Application Error (UAE).
===

Solution

This problem occurs when an MDI child window's window procedure sends a WM_MDISETMENU message to the MDI client window. This usually occurs while the child window is processing the WM_MDIACTIVATE message, when the child is changing the application's current menu to the menu appropriate for that child.

During the processing of the WM_MDIACTIVATE message, Windows modifies its internal structures. Changing the menu at this time is very dangerous. The accepted method is to have the window procedure for the MDI frame window send the WM_MDISETMENU message to the MDI client window.

Therefore, when an MDI child is activated, and it is necessary to change the current menu, the child should call PostMessage to post a

user-defined message to the frame window. When the frame window receives this message, the frame sends a WM_MDISETMENU message to the MDI client window. The frame then redraws the menu bar by calling DrawMenuBar.

Example

To implement the above approach in the MDIDEMO program, perform these three steps:

1. Create a user-defined message. For example:

```
#define IDM_MENUCHANGE    WM_USER + 1000
```

2. When the child window receives a WM_MDIACTIVATE message, instead of sending the MDI client window a WM_MDISETMENU message, use PostMessage to post the new message to the MDI frame window, as follows:

```
PostMessage(hwndFrame, WM_COMMNAND, IDM_MENUCHANGE,
    MAKELONG(hMenuRect, hMenuRectWindow));
```

3. In FrameWndProc, send the WM_MDISETMENU message to the MDI client window in response to the new message. Add this code to the switch statement in processing of the WM_COMMAND message:

```
case IDM_MENUCHANGE:
    SendMessage(hwndClient, WM_MDISETMENU, 0, lParam);
    DrawMenuBar(hwnd);
    return 0;
```

Note that DrawMenuBar must be called to avoid some redrawing problems.

More Information:

For more information on the Windows 3.0 multiple document interface (MDI), query on the following words in the Microsoft Knowledge Base:

prod(winsdk) and MDI

Additional reference words: 3.00 3.0 MSJ V5-4 softlib MDIDEMO.ZIP
KBCategory:
KBSubcategory: UsrMdiMenus

INF: Alternate MDI Tiling Scheme Code Sample Available
Article ID: Q72135

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Software/Data Library contains the source code to a sample program, MDITILE, that demonstrates how to implement an alternate tiling scheme for the multiple document interface (MDI) of Windows version 3.00.

The alternate tiling scheme demonstrated in this sample attempts to tile MDI child windows so they are wider than they are tall. The default MDI tiling scheme sizes the child windows to be tall and narrow. This is unsuitable for MDI applications based on text such as a word processor.

MDITILE can be found in the Software/Data Library by searching on the keyword MDITILE, the Q number of this article, or S13080. MDITILE was archived using the PKware file-compression utility.

More Information:

The horizontally based tiling scheme implemented in this sample uses the following eight-step procedure:

1. Count the number of MDI child windows that are not iconic (represented by an icon). These are the only windows affected by tiling. This value is saved in "cKids."

2. Find the next square (that is, a number "n," such that some integer "x" is the square root, x*x == n), that is greater than the number of windows to tile.

3. Set a column value, "cCols," to "x-1," where "x" is the root of the square found in step 2.

4. Set a row counter, "cRows," to "cKids / cCols."

5. Set a remainder counter, "cMulRows," to "cKids % cCols." This accounts for the child windows that are not included in the base grid of cCols by cRows.

6. Get the dimensions of the client rectangle in which tiling is to occur. Note that it may be desirable to reduce the size of this rectangle to leave space at the bottom of the MDI client window for a row of icons. Divide the vertical height of the rectangle by cRows to get the child window height, and divide the horizontal width of the rectangle by cCols to get the child window width.

7. Loop through each row. In this row, the application will tile cCols child windows. The value "cRows-cMulRows" is the number of rows that can be created with cCols columns. If the rows that can have cCols columns have already been done, then increment cCols so that all remaining rows have an extra window. Since cMulRows are left to tile, adding one window to each row account for all remaining windows.

   This is also why cCols was initially "x-1." If "cCols==x," then the first rows (at the top of the window) would have more windows than rows on the bottom. If this tiling scheme is desired, set cCols to "x" as the assumed number of columns, then decrement cCols when cMulRows have been tiled.

8. Once the number of columns has been calculated, loop through the columns, moving each window to the appropriate section of that row.

Note: It is preferable to move windows with BeginDeferWindowPos(), DeferWindowPos(), and EndDeferWindowPos(), instead of many calls to SetWindowPos(), because the visual effect of the window movement is much cleaner.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiMenus

INF: Suggested Changes to Petzold's MDIDEMO Program

Article ID: Q68119

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

MDIDEMO is the title of a sample Windows program that Charles Petzold introduces in the Windows 3.0 version of his book, "Programming Windows" (Microsoft Press). This article describes three changes that should be made to the code in MDIDEMO, and also to any MDI applications based on MDIDEMO. These changes work around potential problems in the MDIDEMO application.

More Information:

PROBLEM 1
=========

Symptoms

If the user closes a minimized MDI child window, it becomes impossible to close MDIDEMO itself, or to exit the Windows session by closing the Program Manager.

Cause

MDIDEMO will not close until all children of the MDI client window have closed. However, when a minimized MDI child window is closed, the accompanying small title window is hidden but not destroyed. Because this window remains, MDIDEMO is prevented from closing.

Resolution 1

Modify the code to add a line such as the following to the WM_CLOSE case of each child window procedure [in MDIDEMO, the child window procedures are RectWndProc() and HelloWndProc()]:

```
     if (IsIconic(hwnd))
        SendMessage(GetParent(hwnd), WM_MDIRESTORE, hwnd, 0L);
```

This call restores the child window before it is destroyed, and the icon title window is destroyed correctly. MDIDEMO will now close correctly.

Resolution 2

As an alternative, an MDI application's closing procedure can be modeled on the MULTIPAD sample program included with the Windows 3.0 Software Development Kit (SDK). MULTIPAD, with its locally defined CloseAllChildren(), QueryCloseAllChildren(), and QueryCloseChild() procedures, avoids the closing problem.

PROBLEM 2

Symptoms

In MDIDEMO, the MDI child windows do not have any children. If the MDI child windows in an application based on MDIDEMO do have any children, a NULL window handle can be passed to CloseEnumProc(). The application must check to make sure that this NULL window handle is not passed as a parameter to any Windows calls.

Cause

A NULL window handle can be passed to CloseEnumProc() due to an interaction between the order in which EnumChildWindows() enumerates the children of the MDI Client window and the order in which the Client window destroys windows in response to a WM_MDIDESTROY message.

Resolution

Add the following code before the first line in CloseEnumProc():

```
     if (!IsWindow(hwnd))
        return TRUE;
```

PROBLEM 3

Symptoms

The amount of free system resources available after the MDI application is loaded and terminated is lower than the amount of free system resources available before the application is loaded.

Cause

MDIDEMO loads three menus from its resources: hMenuInit, hMenuHello, and hMenuRect. When MDIDEMO exits, the two menus that are not active are not destroyed and the memory occupied by these menus is lost to the system.

Resolution

Add the following code to the FrameWndProc() function:

```
   case WM_DESTROY:
      {
      HMENU hCurrentMenu = GetMenu(hwnd);

      if (hMenuInit != hCurrentMenu)
            DestroyMenu(hMenuInit);
      if (hMenuHello != hCurrentMenu)
            DestroyMenu(hMenuHello);
      if (hMenuRect != hCurrentMenu)
            DestroyMenu(hMenuRect);

      PostQuitMessage(0);
      return 0;
      }
```

Note that DestroyMenu() must NOT be called on the menu that is currently in use, or the program will crash.

For more information on general issues with the Windows 3.0 implementation of MDI, query this knowledge base on the following words:

   prod(winsdk) and 3.00 and MDI

Additional reference words: 3.00 MICS3 R1.2
KBCategory:
KBSubcategory: UsrMdiRare/misc

FIX: No Title for First MDI Child Window Icon
Article ID: Q71897

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012002

SYMPTOMS
In a Windows version 3.0 multiple document interface (MDI) application, if the first MDI child is created minimized (as an icon), the icon is painted but no title window is created or painted. Subsequent MDI child windows are not affected.

RESOLUTION
Microsoft has confirmed this to be a problem in Windows version 3.0. To avoid this problem, create the first MDI child as an icon, then hide and redisplay the window. Note: This approach will cause a flash on the display.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

INF: Multiple Document Interface Enhancements in Windows 3.1

Article ID: Q85010

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The Windows multiple document interface (MDI) provides a way to standardize the user interface of applications that can process more than one "document" simultaneously. Applications that use MDI automatically benefit from improvements in the MDI user interface code within Windows.

The MDI under Windows version 3.1 introduces a number of significant enhancements to the features provided by Windows version 3.0, and it also removes a number of problems. When an application that is designed to use MDI under Windows 3.0 is run on a system equipped with Windows 3.1, the application automatically receives the benefit of many of these enhancements. In addition, an application targeted specifically for Windows 3.1 can use additional new MDI features.

More Information:

The Windows 3.1 MDI corrects a number of problems found in the Windows 3.0 MDI. For a list of problems in Windows 3.0 that have been corrected in Windows 3.1, query on the following words in the Microsoft Knowledge Base:

   buglist3.00 and fixlist3.10 and mdi

The query above provides a list of articles describing the problems with MDI under Windows 3.0; the list also describes methods to work around these problems. Even though an application is no longer required to work around these problems, the methods suggested in each article are compatible with MDI under Windows 3.1.

The "Double-Click" Controversy Finally Comes to a Close

Under Windows 3.0, when the user double-clicks the mouse on the system menu bitmap of a maximized MDI child window, the child window does not close. This behavior caused many complaints when MDI was incorporated into Windows 3.0. A controversy arose regarding this behavior: some believed it to be a design choice; others saw it as a problem with the software.

The design has been changed under Windows 3.1. Using the mouse to double-click the system menu bitmap of a maximized MDI child window closes the child window. The new behavior removes an inconsistency among applications that use the MDI functions built into Windows and two applications that perform their own MDI processing: Microsoft Word

for Windows and Microsoft Excel for Windows.

A Difference with Scroll Bars on MDI Children

Under Windows 3.0, if an application specifies WS_HSCROLL and WS_VSCROLL in the style field of an MDICREATESTRUCT data structure and creates an MDI child window, the scroll bars do not appear. Windows automatically sets the scroll range to zero when it creates the scroll bars. The application must call the SetScrollRange function to specify a positive, non-zero, upper bound for the scroll range to make the scroll bars appear.

Under Windows 3.1, the application behaves somewhat differently. Because Windows 3.1 does not reset the scroll range to zero, the scroll bars appear if the application specifies the WS_HSCROLL and WS_VSCROLL styles. This change might cause problems for an application designed under Windows 3.0 that expected the scroll bars to be hidden until the application explicitly set the scroll range.

The CS_NOCLOSE Class Style

Under Windows 3.0, specifying the CS_NOCLOSE window class style does not change the behavior of MDI child windows. This behavior has been corrected under Windows 3.1. For example, an application specifies CS_NOCLOSE in the style field of a WNDCLASS structure and registers a window class using that structure. If the application creates an MDI child window using the new class, Windows will disable the Close menu option on the child window's system menu.

However, Windows makes other changes to the MDI child's system menu that are not correct. For more information on this problem, query on the following words in the Microsoft Knowledge Base:

   CS_NOCLOSE and MDI

MDI Child Windows No Longer Required to Use Standard Styles

One of the most requested enhancements to Windows MDI provided by Windows 3.1 is the ability to create MDI child windows using any window style. If an application specifies the MDIS_ALLCHILDSTYLES style for its MDI client window, Windows creates the MDI child windows using only the styles specified in the MDICREATESTRUCT data structure. Otherwise, Windows enforces the default MDI child window styles. For example, an application that uses the MDIS_ALLCHILDSTYLES style can create MDI child windows without maximize or minimize buttons.

Multiple Document Interface Message Changes

Windows 3.1 includes several enhancements to MDI system messages. The WM_MDICASCADE and WM_MDITILE messages have been enhanced to accept a cascade flag. If an application sends either of these messages with wParam set to MDITILE_SKIPDISABLED, Windows does not change the position of disabled MDI child windows.

The WM_MDITILE message accepts two additional flags in wParam: MDITILE_HORIZONTAL and MDITILE_VERTICAL. These flags specify that the MDI child windows should be as wide as possible or as tall as possible, respectively.

Under Windows 3.0, the WM_MDINEXT message was not accompanied by any parameters in lParam or wParam. Under Windows 3.1, both parameters are used. The value of wParam specifies the handle of the MDI child window receiving the message. The lParam parameter is a Boolean value specifying whether the next or previous MDI child window is being activated.

The WM_MDISETMENU message also has new functionality. If wParam contains the Boolean value TRUE, the current menus are refreshed. If wParam contains the Boolean value FALSE, the frame menu and window menu are replaced with the menus specified in lParam.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMdi

INF: WM_MDICREATE Message Documented Incorrectly
Article ID: Q71837

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The return value from the WM_MDICREATE message is documented incorrectly on pages 6-75 and 6-76 of the "Windows Software Development Kit Reference Volume 1."

The handle to the newly created child window is returned in the return value low word.

Note: If the child window ID is required and the handle to the child window is available, the GetWindowWord() can be used to obtain the ID.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiRare/misc

INF: Sample Code Demonstrates Superclassing MDI Client Window
Article ID: Q84979

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

An application can use superclassing to add new styles and functionality to a multiple document interface (MDI) client window. SUPERMDI is a file in the Software/Data Library that demonstrates superclassing an MDI client window to change its behavior.

SUPERMDI can be found in the Software/Data Library by searching on the word SUPERMDI, the Q number of this article, or S13440. SUPERMDI was archived using the PKware file-compression utility.

More Information:

Under a variety of circumstances an application might be designed to change the predefined behavior of an MDI client window. Superclassing provides the easiest and cleanest method to change window styles or to handle messages in a unique manner.

By default, an MDI client window cannot accept mouse button double-clicks. If an application superclasses the client window, the application can change the window style of the client window and have it accept double-clicks. An application can also use this method to change the MDI client window background color.

The SUPERMDI sample application demonstrates changing the color of the MDI client window and enabling the MDI client to respond to mouse button double-clicks.

Additional reference words: 3.00 3.10 softlib SUPERMDI.ZIP
KBCategory:
KBSubcategory: UsrMdiRare/misc

FIX: Setting the Focus to the Active MDI Child Window
Article ID: Q69807

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS
   When the CTRL+F6 key combination is used to move from an icon MDI
   child window to a non-iconic MDI child window, the input focus
   remains on the MDI client window.

RESOLUTION
   Microsoft has confirmed this to be a problem in Windows version
   3.0. To work around this problem, you can add code to the window
   procedure for the MDI child window. The code verifies that when a
   non-iconic MDI child window becomes active, it correctly receives
   the input focus. See below for an example of the appropriate code.

   This problem was corrected in Windows version 3.1.

More Information:

In the code below, "hwnd" has been passed as the hWnd parameter of the
WM_MDIACTIVATE message:

```
// if (This window is now the active MDI child
//         && this window is not an icon)
//    {
//    grab the input focus
//    }

case WM_MDIACTIVATE:
   if ((TRUE == wParam) && (!IsIconic(hwnd)))
        {
        SetFocus(hwnd);
        return TRUE;
        }
   else
        break;     // For example, return DefMDIChildProc
```

Note that in many MDI applications, the focus could be set to an
appropriate child of the MDI child, rather than to the MDI child
itself. For example, a text editing program such as MULTIPAD would set
the focus to its child edit control.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiCreatekid

FIX: Destroy Icon MDI Child Window, Title Remains
Article ID: Q71898

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012003

SYMPTOMS
   In a Windows version 3.0 multiple document interface (MDI) application, when a minimized MDI child window is destroyed, the auxiliary window that displays the window title is hidden but is not destroyed.

RESOLUTION
   Microsoft has confirmed this to be a problem in Windows version 3.0. To avoid this problem, Hide the child window and restore it to normal size before sending the WM_MDIDESTROY message to the child window.

   Note: Do NOT destroy the child window while it is minimized, and then subsequently try to destroy the icon title window using the DestroyWindow function, because this may terminate the application or cause Windows to crash.

   This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

INF: Sample Code Demonstrates Window Status Bar
Article ID: Q85203

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application can implement a status bar at the bottom of its main window to provide information to the user. For example, Microsoft Word for Windows and Microsoft Excel for Windows provide status bars. The status bar in these applications presents the state of the CAPS LOCK key, insert mode, extended selection mode, and so forth. STATBAR is a file in the Software/Data Library that demonstrates how to create a status bar for an application.

STATBAR can be found in the Software/Data Library by searching on the word STATBAR, the Q number of this article, or S13441. STATBAR was archived using the PKware file-compression utility.

More Information:

The STATBAR application creates the status bar when its main window receives a WM_CREATE message. Each time the main window receives a WM_KEYUP message, the application determines whether the released key is a "toggle key," such as the CAPS LOCK, NUM LOCK, or INS key. If so, the application draws the status bar to reflect the current state of the toggle key that changed state. When the status bar receives a WM_PAINT message, the status bar repaints its entire client area.

Additional reference words: 3.00 3.10 softlib STATBAR.ZIP
KBCategory:
KBSubcategory: UsrMdiToolbars

BUG: System Menu Wrong for CS_NOCLOSE-Style MDI Child
Article ID: Q85331

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

SYMPTOMS
========

When an application registers a window class with the CS_NOCLOSE style and creates a multiple document interface (MDI) child window using the class, the menu items in the child window's system menu are incorrect. Specifically, the Switch To option appears where the Next option belongs. Choosing Switch To activates the Windows Task Manager.

STATUS
======

Microsoft has confirmed this to be a problem in Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMdiMenus

Using MoveWindow() to Move an Iconic MDI Child and Its Title

Article ID: Q70079

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Although Windows version 3.00 does not have API calls to support dragging iconic windows (windows represented by an icon), calls to MoveWindow() can be used to implement a dragging functionality for iconic multiple-document interface (MDI) child windows.

However, the icon's title is not in the same window as the icon, so when the icon is moved, the title will not automatically move with it. The title is in a small window of its own, so a separate call to MoveWindow() must be made to place the title window correctly below the icon.

The information below describes how to get the window handle for the small title window, so that the appropriate call to MoveWindow() can be made.

This information applies to applications created with the Windows Software Development Kit (SDK) version 3.00.

More Information:

The icon's title window is a window of class #32722. This window is a child of the MDI client window, and its owner is the icon window.

To get the window handle to the appropriate icon title window for a given icon, enumerate all the children of the MDI client window, looking for a window whose owner is the icon.

You can use EnumChildWindows() to loop through all the children of the MDI client window.

You can use GetWindow(..., GW_OWNER) to check the parent of each window.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdi

Creating a Hidden MDI Child Window

Article ID: Q70080

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Whenever Windows version 3.0 creates a new multiple-document interface (MDI) child window in response to a WM_MDICREATE message, it makes that child window visible.

The information below describes how to create a hidden MDI child window without causing an unattractive "flash" on the screen as the window is created visible and then hidden. This information applies to applications created with the Windows Software Development Kit (SDK) version 3.0.

More Information:

A code fragment such as the following can be used to create an invisible MDI child:

```
MDICREATESTRUCT mcs;            // structure to pass with WM_MDICREATE
HWND            hWndMDIClient;  // the MDI client window
HWND            hwnd;           // temporary window handle

      ...

// assume that we have already filled out the MDICREATESTRUCT...

// turn off redrawing in the MDI client window
SendMessage(hwndMDIClient, WM_SETREDRAW, FALSE, 0L);

/*
 * Create the MDI child. It will be created visible, but will not
 * be seen because redrawing to the MDI client has been disabled
*/
hwnd = (WORD)SendMessage(hwndMDIClient,
                         WM_MDICREATE,
                         0,
                         (LONG)(LPMDICREATESTRUCT)&mcs);

// hide the child
ShowWindow(hwnd, SW_HIDE);

// turn redrawing in the MDI client back on,
// and force an immediate update
SendMessage(hwndMDIClient, WM_SETREDRAW, TRUE, 0L);
InvalidateRect( hwndMDIClient, NULL, TRUE );
UpdateWindow( hwndMDIClient );
      ...
```

For more information on the Windows 3.0 MDI, query on the following words:

   prod(winsdk) and MDI

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiCreatekid

INF: Minimal MDI Application Source in Software Library
Article ID: Q71660

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

BLANDMDI is a source code sample in the Software/Data Library that demonstrates a minimal Windows multiple document interface (MDI) application. BLANDMDI can be found in the Software/Data Library by searching on the word BLANDMDI, the Q number of this article, or S13021. BLANDMDI was archived using the PKware file-compression utility.

The rest of this article discusses some further sources of information regarding Windows version 3.00 MDI programming.

More Information:

Sources of Information on MDI Programming

Chapter 21 of the "Windows Software Development Kit Guide to Programming" provides information on developing Windows version 3.00 MDI applications.

The Windows Software Development Kit (SDK), version 3.00, contains a sample MDI application called MULTIPAD. BLANDMDI was created by modifying MULTIPAD to remove all code not essential to an MDI application.

For more information on MDI, query on the following words:

   prod(winsdk) and MDI

MDIDEMO

Another source of information on Windows MDI is Chapter 18 of "Programming Windows," second edition, by Charles Petzold (Microsoft Press). This chapter includes a sample program called MDIDEMO. Developers who are using MDIDEMO as a base for their MDI application should query in this knowledge base for more information on this program:

   prod(winsdk) and MDIDEMO

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdi

INF: Preventing an MDI Child Window from Changing Size
Article ID: Q71669

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In an application using the Windows version 3.00 multiple document interface (MDI), MDI child windows are always created with a maximize button, a minimize button, and a thick "sizing" border. Currently, it is not possible to modify this default behavior.

However, it is possible to prevent an MDI child from being maximized, minimized, or sized. This can be done by performing two steps:

1. Disable the Maximize, Minimize, and Size options on the child window's system menu.

2. Trap the appropriate WM_SYSCOMMAND messages in the MDI child window's window procedure.

This article contains a sample MDI child window procedure that illustrates this technique.

More Information:

The following sample MDI child window procedure can be used to implement an MDI child that can not be maximized, minimized or sized:

```
LONG FAR PASCAL SampleMDIChildWndProc(hWnd, msg, wParam, lParam)

HWND    hWnd;
WORD    msg;
WORD    wParam;
LONG    lParam;

// This sample MDI child window procedure can be used to prevent the
// MDI child from being maximized, minimized, or sized.
//
// Note that the child will still have a maximize button, a minimize
// button, and a thick "sizing" border. Currently, there is no way to
// prevent Windows from creating MDI children with these default
// styles.
//
// We can, however, disable the maximizing, minimizing, and sizing
// functionality by:
//
// 1. Disabling system menu options
// 2. Trapping WM_SYSCOMMAND messages

{
    HMENU    hSystemMenu;    // the MDI child's system menu
```

```
    switch (msg)
        {
    case WM_INITMENU:
        // Disable and gray the Maximize, Minimize, and Size items
        // on the MDI child's system menu
        hSystemMenu = GetSystemMenu(hWnd, FALSE);
        EnableMenuItem(hSystemMenu, SC_MAXIMIZE, MF_GRAYED |
             MF_BYCOMMAND);
        EnableMenuItem(hSystemMenu, SC_MINIMIZE, MF_GRAYED |
             MF_BYCOMMAND);
        EnableMenuItem(hSystemMenu, SC_SIZE, MF_GRAYED |
             MF_BYCOMMAND);
        break;

    case WM_SYSCOMMAND:
        // "Eat" these system commands to disable maximizing,
        // minimizing, and sizing functionality.
        // Note that wParam must be combined with 0xFFF0, using
        // the C bitwise AND (&) operator, when processing
        // the WM_SYSCOMMAND message.
        switch (wParam & 0xFFF0)
            {
        case SC_MAXIMIZE:
        case SC_MINIMIZE:
        case SC_SIZE:
             return 1L;

        default:
             break;
            }
        break;

    default:
        break;
        }

    return DefMDIChildProc(hWnd, msg, wParam, lParam);
}
```

For more information on Windows MDI, query on the following words:

   prod(winsdk) and MDI

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdi

FIX: MDI Display Bad When Application Restored From Icon
Article ID: Q71899

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012004

SYMPTOMS
When a Windows 3.0 multiple document interface (MDI) application is started as an icon, iconic child windows are not painted properly.

RESOLUTION
Microsoft has confirmed this to be a problem in Windows version 3.0. To avoid this problem, perform the following four steps:

1. Create the MDI frame window off of the screen with WS_VISIBLE as one of the window styles.

2. Hide the window by calling ShowWindow with SW_HIDE.

3. Move the hidden frame window on to the desired starting position of the screen.

4. Show the window at the bottom of the screen by calling ShowWindow with SW_SHOWMINIMIZED.

This problem was corrected in Windows version 3.1.

More Information:

It is common for MDI applications to create iconic child windows when the MDI frame window processes the WM_CREATE message. If an application that uses this technique is initially displayed as an icon, the application does not behave as expected after it is restored to normal size. When the MDI application is restored, the MDI client's vertical scroll bar is shown without any of the MDI children appearing on the screen. The thumb on the vertical scroll bar is placed at the bottom. When the thumb is moved to the top of the scroll bar, the title windows for the MDI children are visible; however, the icons are not.

The following code fragment implements the resolution detailed above:

```
/* Create the frame window off of the screen with WS_VISIBLE. */
hwndFrame = CreateWindow(szFrame, szWindowName,
              WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_VISIBLE,
              -X_LENGTH, -Y_LENGTH, 0, 0,
              NULL, NULL, hInst, NULL);
if (!hwndFrame)
```

```
      return FALSE;

    /* Move the frame window on to the screen hidden. */
    ShowWindow(hwndFrame, SW_HIDE);
    MoveWindow(hwndFrame, X_START, Y_START, X_LENGTH, Y_LENGTH, TRUE);

    /* Show the MDI application minimized. */
    ShowWindow(hwndFrame, SW_SHOWMINIMIZED);
    UpdateWindow(hwndFrame);
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

FIX: Creating Zoomed MDI Child w/ Scroll Bar UAEs

Article ID: Q71900

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012005

SYMPTOMS

Under enhanced mode Windows version 3.0, when a multiple document interface (MDI) application creates a maximized MDI child window with at least one visible scroll bar, the application terminates with an unrecoverable application error (UAE). When Windows is running in standard or real mode, Windows reports a fatal exit 0x00FF.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. To avoid this problem, specify the desired scroll bar style(s) when the MDI child window is created. Then set the scroll range for the scroll bar(s) AFTER processing of the WM_MDICREATE message has completed.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

PRWIN9012006: New Zoomed MDI Child Over Zoomed Child Flashes
Article ID: Q72025

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012006

SYMPTOMS

In a Windows version 3.00 multiple document interface (MDI) application, creating a new maximized MDI child when the currently active MDI child is maximized causes visually unattractive multiple repaints.

CAUSE

The following list summarizes what happens in this situation:

1. The active MDI child is restored to normal size and repainted.
2. The frame window's menu bar is repainted.
3. The new maximized MDI child is created and painted.
4. The frame window's menu bar is painted a second time.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.00. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

FIX: Activate Next MDI Child, Present Child Zoomed

Article ID: Q72026

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012007

SYMPTOMS

In a Windows version 3.0 multiple document interface (MDI) application, when the currently active MDI child is maximized, using the MDI system accelerator CTRL+F6 to activate the next MDI child may leave options in the newly activated child's system menu inappropriately disabled.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. To avoid this problem, modify the MDI child window procedure to process WM_MDIACTIVATE messages. When one of these messages is received, check to see if the child is being activated (wParam != 0) and is currently maximized (the IsZoomed function returns TRUE). If so, use the GetSystemMenu function to retrieve the handle to the child window's system menu and then enable/disable menu items as appropriate.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

FIX: Windows Overrides MDI Child Window Styles

Article ID: Q72027

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012008

SYMPTOMS

Most user-specified styles used in creating a Windows version 3.0 multiple document interface (MDI) child window are overridden by MDI default styles. Specifically, before the child window is created, all user-specified window styles except the WS_MAXIMIZE, WS_MINIMIZE, WS_VSCROLL, WS_HSCROLL, and WS_CLIPCHILDREN styles are removed and the WS_CHILD, WS_CLIPSIBLINGS, WS_SYSMENU, WS_CAPTION, WS_THICKFRAME, WS_MAXIMIZEBOX, and WS_MINIMIZEBOX styles are added.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. No reliable way exists to modify the default behavior. Direct manipulation of a window's style bits after the window is initially created can cause unpredictable behavior and should not be attempted.

This problem was corrected in Windows version 3.1. An application can specify the MDIS_ALLCHILDSTYLES style for the MDI client window, which prevents Windows from modifying the style of MDI child windows. Modifying Windows in this manner provides backward compatibility for MDI applications developed under Windows 3.0.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

FIX: Double-Click Zoomed MDI Child System Menu NOP

Article ID: Q72028

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012009

SYMPTOMS

In a Windows version 3.0 multiple document interface (MDI) application, double-clicking the system menu of a maximized MDI child window does not cause the child to close.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. To work around this problem, process WM_NCLBUTTONDBLCLK messages in the window procedure for the MDI frame window. When a message is received, send the MDI client window a WM_MDIGETACTIVE message to determine if the child window is maximized. If it is, and if the double-click occurred over the child's system menu icon, send the child a WM_CLOSE message.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

FIX: MDI Child Windows Ignore CS_NOCLOSE Style
Article ID: Q72029

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9012010

SYMPTOMS
   In a Windows version 3.0 multiple document interface (MDI)
   application, the window class style CS_NOCLOSE does not affect MDI
   child windows.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiBugs

BUG: MDI More Windows Dialog Activates Wrong Child

Article ID: Q86412

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows
  version 3.1

SYMPTOMS
========

In a multiple document interface (MDI) application developed for Microsoft Windows version 3.1, when the application creates more than nine MDI child windows, the user chooses More Windows from the Window menu, and selects an MDI child window to activate from the list displayed in the Select Window dialog box, the application activates a different window.

CAUSE
=====

When one or more MDI child windows are hidden by an application, the default frame window procedure (DefFrameProc) activates the wrong window.

STATUS
======

Microsoft has confirmed this to be a problem in Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMdiBugs

PRB: MDI Program Menu Items Changed Unexpectedly

Article ID: Q74789

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

In an application for the Microsoft Windows graphical environment developed using the Windows multiple document interface (MDI), when the Window menu changes to indicate the addition of MDI child windows, another menu of the application is also changed.

For example, if the menu resembles the following before any windows are opened

```
FILE             WINDOW
   Load             Cascade
   Save             Tile
   Save As...       Arrange Icons
   Exit
```

the menu might resemble the following after the first window is opened:

```
FILE             WINDOW
   Load             Cascade
   1: MENU.TXT      Tile
   Save As...       Arrange Icons
   Exit             -------------
                    1: MENU.TXT
```


CAUSE

One or more menu items have menu-item identifiers that are greater than or equal to the value of the idFirstChild member of the CLIENTCREATESTRUCT data structure used to create the MDI client window.

RESOLUTION

Change the value of the idFirstChild member to be larger than any menu item identifiers.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMdiMenus

INF: Changing the Default Background Color of an MDI Client
Article ID: Q75002

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Normally, a multiple-document interface (MDI) client window defines its background color to be the system color value COLOR_BACKGROUND+1, which is defined in the Windows Software Development Kit (SDK) file WINDOWS.H. This value corresponds to the window background color defined by the user in the Control Panel.

It is possible to alter the default background color of an MDI client window by subclassing the MDI client window procedure, and processing the WM_ERASEBKGND message. All other messages are passed to the original MDI client window procedure.

More Information:

During processing of the WM_ERASEBKGND message, the class word that defines the MDI client's background color is altered using the SetClassWord() function. Then, control is passed to the original MDI client window procedure to paint and update the window. When control returns from the original MDI client window procedure, the MDI client's class word that defines the background color is set back to its original value so that other MDI applications currently running are not affected.

Below is a sample window procedure that can be used to subclass the default MDI client window procedure.

```
LONG FAR PASCAL MdiClientWndProc(hwnd, msg, wParam, lParam)
HWND hwnd;
WORD msg, wParam;
LONG lParam;
{
  LONG lRetCode;

  switch (msg)
    {
  case WM_ERASEBKGND:
    {
    HBRUSH hbrNew, hbrOld;
                                       // create a solid purple brush
    hbrNew = CreateSolidBrush(RGB(75, 45, 145));
                                    // set the class background color
    hbrOld = SetClassWord(hwnd, GCW_HBRBACKGROUND, hbrNew);
                   // let the old window procedure handle the message
    lRetCode = CallWindowProc(lpfnMDIClientOld, hwnd, msg, wParam,
                lParam);
```

```
                             // set background back to original value
    SetClassWord(hwnd, GCW_HBRBACKGROUND, hbrOld);
    DeleteObject(hbrNew);         // all done with this, so delete it
    break;
    }

  default:
    lRetCode = CallWindowProc(lpfnMDIClientOld, hwnd, msg, wParam,
                lParam);
    break;
    }
  return lRetCode;
}
```

This code fragment is taken from MDISUBCL.ZIP, a file in the Software Library, which contains an application that demonstrates this operation.

MDISUBCL can be found in the Software/Data Library by searching on the keyword MDISUBCL, the Q number of this article, or S13131. MDISUBCL was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiRare/misc

PRB: No Scroll Bars Displayed in an MDI Child Window

Article ID: Q76533

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS
When a Windows multiple document interface (MDI) child window is created and the style field of the MDICREATESTRUCT contains WS_HSCROLL and/or WS_VSCROLL, no scroll bar is displayed.

CAUSE
The scroll bar scroll range has not been set.

RESOLUTION
Set the scroll range to a nonzero interval.

More Information:

The scroll range can be set during the processing of the WM_CREATE message or in response to a user action when the object to be displayed extends beyond the window. The following code sets the scroll range:

```
// hWnd is the handle to the MDI child window.
// 0 is the minimum scrolling position
// 100 is the maximum scrolling position

// set horizontal scroll bar range.
SetScrollRange(hWnd, SB_HORZ, 0, 100, TRUE);

// set vertical scroll bar range.
SetScrollRange(hWnd, SB_VERT, 0, 100, TRUE);
```

The scroll bar scroll range can be set to any nonzero interval provided that the difference between the maximum scrolling position and the minimum scrolling position is not greater than 32,767.

Note: There is a problem in Windows 3.0 that causes an unrecoverable application error (UAE) if an MDI child window with scroll bars has the WS_MAXIMIZE style. For more information, query on the words:

prod(winsdk) and ws_maximize

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrMdiBugs

FIX: Maximized MDI Child with CS_NOCLOSE Problems
Article ID: Q77474

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9110010

SYMPTOMS
   If an MDI child window is created with the WS_MAXIMIZE style and
   the CS_NOCLOSE style, two system menus and two window restore
   buttons are added to the application's menu bar. Closing the
   application causes an unrecoverable application error (UAE).

RESOLUTION
   To avoid this problem, create the MDI child window, then send it a
   WM_MDIMAXIMIZE message.

   Microsoft has confirmed this to be a problem in Windows version
   3.0. We are researching this problem and will post new information
   here as it becomes available.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMdiMenus

PRB: Processing the WM_QUERYOPEN Message in an MDI Application
Article ID: Q99411

Summary:

SYMPTOMS

When a multiple document interface (MDI) child window processes the WM_QUERYOPEN message to prevent the child from being restored out of a minimized state, the system menu and restore button are not removed from the menu bar when a maximized MDI child loses the focus or is closed.

This problem occurs only when the maximized MDI child loses the focus or is closed and the focus is given to an MDI child that returns 0 (zero) on the WM_QUERYOPEN message.

CAUSE

When an MDI child is maximized, the system menu and restore button are added to the frame menu as bitmap menu items. When a maximized MDI child is destroyed or another MDI child is given the focus, the MDI child given the focus afterwards is maximized to replace the old MDI child. Windows cannot maximize an MDI child when it is processing the WM_QUERYOPEN message, and therefore the child is not maximized. Unfortunately, the system menu and restore button bitmaps are not removed from the menu bar.

RESOLUTION

To prevent this problem, restore the maximized MDI child before giving the focus to another child.

More Information:

It may sometimes be desirable to prevent an MDI child from being restored during part or all of its life. This can be done by trapping the WM_QUERYOPEN message by placing the following code in the window procedure of the MDI child:

```
   case WM_QUERYOPEN:
      return 0;
```

Unfortunately, this causes the added restore and system menu bitmaps to remain on the menu bar when a maximized MDI child loses the focus or is closed and the focus is given to a child processing this message. The following code can be used to restore a maximized MDI child when it loses the focus:

```
      case WM_MDIACTIVATE:
         if ((wParam == FALSE) && (IsZoomed(hwnd)))
            SendMessage(hwndMDIClient, WM_MDIRESTORE, hwnd, 0L);

         return DefMDIChildProc (hwnd, msg, wParam, lParam);
```

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMdiCreatekid

PRB: Pressing the ENTER Key in an MDI Application

Article ID: Q99799

Summary:

SYMPTOMS

In a standard Microsoft Windows version 3.1 multiple document interface (MDI) application, when a minimized MDI child is active and the user presses the ENTER key, the child is not restored.

This is inconsistent with other MDI applications, such as File Manager and Program Manager. An MDI child in one of these applications is restored when it is the active MDI child and the ENTER key is pressed. When a normal Windows application is minimized and the user presses ENTER, that application is restored to a normal state.

RESOLUTION

One quick workaround to this problem is to create an accelerator for the ENTER key and restore the minimized MDI child when the key is pressed.

More Information:

It may be desirable to implement the same restore feature that File Manager and Program Manager have implemented in order to enable the user to restore an MDI child by pressing the ENTER key. If this feature is implemented, then the MDI application can be consistent with other popular applications such as File Manager, Microsoft Excel, and Microsoft Word.

To achieve this effect in an MDI application, create an accelerator in the accelerator table of the resource file for the application. This can be done as follows:

```
MdiAccelTable ACCELERATORS
     {
     .  .  .
     .  .  .
     VK_RETURN, IDM_RESTORE, VIRTKEY
     }
```

After this accelerator has been installed in the MDI application, each time the ENTER key is pressed by the user, an IDM_RESTORE command will be sent to the MDI frame window's window procedure through a WM_COMMAND message. When the MDI frame receieves this message, its window procedure should retrieve a handle to the active MDI child and determine if it is minimized. If it is minimized, then it can restore the MDI child by sending the MDI client a WM_MDIRESTORE message. This can all be done with the following code:

```
        case IDM_RESTORE:
           {
           HWND hwndActive;
```

```
            hwndActive = SendMessage(hwndClient,WM_MDIGETACTIVE,0,0L);
            if (IsIconic(hwndActive))
               SendMessage(hwndClient,WM_MDIRESTORE,hwndActive,0L);
            break;
            }
```

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMdiMdiclient

SAMPLE: Customizing the MDI Window Menu

Article ID: Q99803

Summary:

The MDIWINMN sample in the Software/Data Library demonstrates how to customize the Window menu of a multiple document interface (MDI) frame window. MDIWINMN shows how to limit the number of listed MDI children in the Window menu to any number and how to bring up a customized dialog box when the More Windows menu item is selected. This sample also shows how to list all the MDI children in the Window menu without using a More Windows menu item.

MDIWINMN can be found in the Software/Data Library by searching on the word MDIWINMN, the Q number of this article, or S14184. MDIWINMN was archived using the PKware file-compression utility.

More Information:

To compile, type NMAKE to build the sample; this sample limits the number of MDI child windows listed in the Window menu and brings up the customized dialog box when the More Windows menu item is selected. Type NMAKE MENU=NOLIMIT to build the sample that lists all the the MDI children in the Window menu without using a More Windows menu item.

The following code changes must be made in the window procedure of the MDI frame window to customize the Window menu:

1. Specify NULL as the Window menu handle when creating the MDI client, because the application handles all the processing of the Window menu:

```
CLIENTCREATESTRUCT ccs;
ccs.hWindowMenu  = NULL;
ccs.idFirstChild = IDM_FIRSTCHILD;
```

2. Update the Window menu using ChangeWindowMenu when the frame window receives WM_INITMENUPOPUP. ChangeWindowMenu is implemented in the sample code:

```
case WM_INITMENUPOPUP:
     if ((HMENU)wParam == hMenuWindow)
        ChangeWindowMenu(hMenuWindow);
     break;
```

3. Call ProcessWindowMenu in the default processing of WM_COMMAND received by the frame window. ProcessWindowMenu is implemented in the sample code:

```
case WM_COMMAND:
 {
     switch (wParam)
     {
          :
          :
```

```
         default:
            if (ProcessWindowMenu(wParam))
                  return 0L;
            else return DefFrameProc(hwnd, g_hwndMDIClient,
                                    msg, wParam, lParam);
         }
      }
```

ChangeWindowMenu removes the old listing of MDI children from the Window menu. It then enumerates the MDI child windows and adds their names to the Window menu so the Window menu reflects the current state when shown.

ProcessWindowMenu processes WM_COMMAND messages that result from menu selections of MDI children in the Window menu. It also brings up a customized dialog box when the More Windows menu item is selected.

Additional reference words: MDIWINMN.ZIP morewindows
KBCategory:
KBSubcategory: UsrMdiMenus

BUG: Iconic MDI Application Titles Do Not Update Properly

Article ID: Q104137

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1

SYMPTOMS
========

In a Windows 3.1 multiple document interface (MDI) application, the icon title of the MDI frame window does not update when the SetWindowText function is used to change the text of the icon title.

CAUSE
=====

In an MDI application, the main window calls DefFrameProc instead of DefWindowProc to handle unprocessed messages. DefFrameProc does not handle the WM_SETTEXT message (which is generated by SetWindowText) correctly when the main window is minimized, because it does not update the title after the change has been made.

RESOLUTION
==========

To update the minimized MDI frame's title, you can process the MDI frame's WM_SETTEXT message and call both the DefFrameProc and DefWindowProc functions. The following code demonstrates this workaround:

```
   case WM_SETTEXT:
       DefFrameProc (hwnd,hwndMDIClient,msg,wParam,lParam);
       return DefWindowProc (hwnd,msg,wParam,lParam);
```

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMdiDefFrame

INF: Inserting Right Justified Text in a Menu in Windows 3.0

Article ID: Q67063

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The "\a" character used with the InsertMenu() function is NOT considered to be a C language "\a", but is instead considered to be a resource compiler "\a". This code in the resource compiler is translated to a backspace character. Thus, an "\a" used in the resource compiler is really considered to be a "\b" or an "\x08" within the C Language.

When an "\x08" (backspace) is used, the menu item behaves as expected. However, you must place at least one character, which can be a space, before the "\b". For example:

```
   (LPSTR)" \bsometext"
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Adding Items to the System Menus of All Applications
Article ID: Q74695

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

ADMENU.ZIP is a code sample on the Software/Data library that demonstrates adding new menu items to the system menus of all applications running under Windows at a given time.

ADMENU.ZIP can be found in the Software/Data Library by searching on the keyword ADMENU, the Q number of this article, or S13151. ADMENU was archived using the PKware file-compression utility.

More Information:

ADMENUAPP adds two menu items to the system menu of each application running under Windows. To do this, it uses two system-wide hooks: a WH_GETMESSAGE hook and a WH_CALLWNDPROC hook. Both of these hooks are implemented in ADMENULIB, a fixed-code DLL.

Whenever the WH_CALLWNDPROC hook function sees a WM_INITMENU message being sent to a top-level window, it appends the two new menu items to that window's system menu. The items are removed when the menu is closed.

The two menu items that are added appear as "New Menu Item 1" and "New Menu Item 2." If the user selects one of these items, the WH_GETMESSAGE hook procedure sees the resulting WM_COMMAND message and responds by posting a private message to ADMENUAPP. ADMENUAPP handles the private message by displaying a dialog box stating that one of the two added items was selected.

For more information on adding menu items to another application's system menu, query on the following words:

   prod(winsdk) and SUBAPP

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenSysmenu

INF: Customizing a Pop-Up Menu

Article ID: Q12118

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following are three methods to customize a pop-up menu:

1. Use the word SEPARATOR in a pop-up menu, to produce a horizontal
   bar.

2. Use the word MENUBREAK, to start the menus on another column.

3. Place the vertical bar symbol in the menu string to display a
   vertical bar on the menu.

Additional reference words: 2.00 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Changing How Pop-Up Menus Respond to Mouse Actions

Article ID: Q65256

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The TrackPopupMenu function allows an application to receive input from a menu that is displayed anywhere within the application's client area. This article demonstrates how to change the menu's default behavior for mouse selections.

More Information:

The default action for floating pop-up menus maintained with TrackPopupMenu is as follows:

1. If the menu is displayed in response to a keystroke, the pop-up menu is visible until the user selects a menu item or presses ESC.

2. If the menu is displayed in response to a WM_*BUTTONUP message, it acts as if it were displayed in response to a keystroke.

Note: In the context of this article, the * in the WM_*BUTTONUP and WM_*BUTTONDOWN messages can be L (left mouse button), M (middle mouse button), or R (right mouse button).

An application can change the behavior of a floating pop-up menu displayed in response to a WM_*BUTTONDOWN message to keep it visible after the mouse button is released. However, when an application uses the techniques described below, it changes the menu's user interface. Specifically, to change menu selections with the mouse, the user must first release the mouse button and then press it again. Dragging the mouse between items with the button down, without releasing the button at least once, will not change the selection.

To cause a floating pop-up menu to remain visible after it is displayed in response to a WM_*BUTTONDOWN message, follow these four steps:

1. In the application, allocate a 256 byte buffer to hold the key state.

2. Call the GetKeyboardState function with a far pointer to the buffer to retrieve the keyboard state.

3. Set the keyboard state for the VK_*BUTTON index in the keyboard state array to 0.

4. Call SetKeyboardState with a far pointer to the buffer to register the change with Windows.

The keyboard state array is 256 bytes. Each byte represents the state of a particular virtual key. The value 0 indicates that the key is up, and the value 1 indicates that the key is down. The array is indexed by the VK_ values listed in Appendix A of the "Microsoft Windows Software Development Kit Reference Volume 2" for version 3.0.

The code fragment below changes the state of the VK_LBUTTON to 0 (up) during the processing of a WM_LBUTTONDOWN message. This causes TrackPopupMenu to act as if the menu were displayed as the result of a WM_LBUTTONUP message or of a keystroke. Therefore, the menu remains visible even after the mouse button is released and the WM_LBUTTONUP message is received. Items on this menu can be selected with the mouse or the keyboard.

```
   switch (iMessage)
      {
   case WM_LBUTTONDOWN:
      static BYTE rgbKeyState[256];

      GetKeyboardState(rgbKeyState);
      rgbKeyState[VK_LBUTTON] = 0;      // 0==UP, 1==DOWN
      SetKeyboardState(rgbKeyState);

      // Create the pop-up menu and call TrackPopupMenu.
      break;
      }
```

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMenTrackpop

INF: How to Create a Menu-Bar Item Dynamically
Article ID: Q21568

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Sometimes it may be desirable to add and delete menus within an application dynamically, rather than defining the menu in the .RC file.

Menus can be created, altered, and deleted using the following functions:

1. CreateMenu() creates an empty menu (menu bar), which can then be added to.

2. ChangeMenu() is used to append, insert, delete, and make other changes to the menu (menu bar) and/or its pop-up menus.

3. SetMenu() sets a given window's menu to the one specified.

4. DrawMenuBar() redraws the menu for the specified window.

Therefore, use CreateMenu() to create the menu, use ChangeMenu() to add menu items, use SetMenu() to attach this menu to the window, and call DrawMenuBar() to display the menu.

Note: Remember that when SetMenu() is used, the new menu replaces the menu that is already there. The following must be done or system resources will decrease because the menu will not be freed when the application exits:

1. Use GetMenu() to retrieve the current menu.

2. SetMenu() to the new menu.

3. Destroy the old menu.

More Information:

The following code example should be put in the sample Hello application in the HelloWndProc procedure (add to HELLO.C):

```
#define HI_MENU     100
#define THERE_MENU  101

 BOOL  added_hi = FALSE;
 BOOL  added_there = FALSE;
 HMENU hMenu;
```

```
 case WM_CHAR:
     if (!added_hi) {
       hMenu = CreateMenu();
       ChangeMenu( hMenu, NULL, (LPSTR)"Hi", HI_MENU, MF_APPEND );
       SetMenu( hWnd, hMenu);
       DrawMenuBar(hWnd);
       added_hi = TRUE;
     }
     break;
 case WM_COMMAND:
      switch (wParam) {
        case HI_MENU:
         if (!added_there) {
             hMenu = GetMenu(hWnd);
             ChangeMenu( hMenu, 1, (LPSTR)"There", THERE_MENU, MF_APPEND;
             DrawMenuBar(hWnd);
             added_there = TRUE;
          }
          break;
        case THERE_MENU:
          MessageBox (hWnd, (LPSTR)"no more menus allowed",
                           (LPSTR)"Menu Test", MB_OK );
          break;
        default:
          return(DefWindowProc(hWnd, message, wParam, lParam));
          break;
      }
      break;
```

When a key is pressed, a WM_CHAR message will appear and will add a menu item to the main menu bar. When the "Hi" menu item is clicked, a WM_COMMAND message will appear that will then generate another menu item called "There." Clicking the "There" menu item will cause a message box to appear.

Additional reference words: 2.x 2.00 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrMenCreatemen

PRB: MENUITEMTEMPLATE Structure Is Documented Incorrectly
Article ID: Q66247

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

The MENUITEMTEMPLATE structure is documented incorrectly on page 7-50 of the "Microsoft Windows Software Development Kit Reference Volume 2," version 3.0. The documentation incorrectly defines the MENUITEMTEMPLATE as follows:

```
typedef struct
    {
    WORD    mtOption;
    WORD    mtID;
    LPSTR   mtString;
    } MENUITEMTEMPLATE;
```

STATUS

The correct structure is found in WINDOWS.H, which declares the structure as follows:

```
typedef struct
    {
    WORD    mtOption;
    WORD    mtID;
    char    mtString[1];
    } MENUITEMTEMPLATE;
```

Single-item arrays, such as mtString, provide a named field to use to access memory. The actual text of the string is stored in the structure, not a pointer to text stored elsewhere.

The structure is documented correctly in the "Microsoft Windows Software Development Kit: Programmer's Reference Volume 3: Messages, Structures, and Macros," for version 3.1 on page 318.

More Information:

MENUTEMP is a sample program in the Software/Data Library that demonstrates using the MENUITEMTEMPLATE structure and the LoadMenuIndirect function. MENUTEMP can be found in the Software/Data Library by searching on the word MENUTEMP, the Q number of this article, or S12832. MENUTEMP was archived using the PKware file-compression utility.

The declaration of MENUITEMTEMPLATE in WINDOWS.H is correct. If a program attempts to assign an LPSTR to mtString, the C compiler generates an error. Listed below is an erroneous code sample:

```
   MENUITEMTEMPLATE    mit;
   LPSTR               lpch;
   ...
   mit.mtString = lpch;
   ...
```

The mtString field is a 1-byte placeholder for the array. Because a LPSTR is 4 bytes long, it cannot be assigned to a 1-byte quantity.

The mtString[1] declaration in the structure serves as a placeholder for an arbitrary number of characters. An application that uses the MENUITEMTEMPLATE structure must allocate memory both for the template itself and the string that is copied into mtString.

The following code sample demonstrates how an application might create a MENUITEMTEMPLATE structure for a checked menu item having an ID value of 100 and "&Menuitem" as its text:

```
    HANDLE              hMem;
    LPMENUITEMTEMPLATE  lpmit;
    static char         szMenuItem[] = "&Menuitem";

    ...

    // Note that the single char in the MENUITEMTEMPLATE structure
    // provides space for the null terminator on the string.
    hMem = LocalAlloc(LMEM_MOVEABLE, sizeof(MENUITEMTEMPLATE)
                                        + lstrlen(szMenuItem));

    // LocalLock function returns a near pointer;
    // no problem casting to a far pointer
    lpmit = (LPMENUITEMTEMPLATE)LocalLock(hMem);

    // Set the ID and the checked flag.
    lpmit->mtOption = MF_CHECKED;
    lpmit->mtID = 100;

    // Copy the menu item text.
    lstrcpy(lpmit->mtString, szMenuItem);

    ...

    // Make the following call, when a pointer is no longer needed.
    LocalUnlock(hMem);

    ...

    // Make the following call, when the MENUITEMTEMPLATE
    // is no longer needed.
    LocalFree(hMem);

    ...
```

Additional reference words: 3.00 softlib MENUTEMP.ZIP
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Sample Code Demonstrates Using TrackPopupMenu
Article ID: Q80225

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Under Windows version 3.1, the TrackPopupMenu function documents and uses its last parameter. This parameter is an LPRECT, which describes the rectangle in which the user can release the mouse button without dismissing the menu.

This newly documented parameter is useful for applications that display a pop-up menu in response to a WM_LBUTTONDOWN message. When using this parameter, it is not necessary to hold down the mouse button until a menu selection is made. The user can release the button and select an item from the menu with a second button click.

TRACKPOP is a file in the Software/Data Library that demonstrates using the TrackPopupMenu function. TRACKPOP can be found in the Software/Data Library by searching on the word TRACKPOP, the Q number of this article, or S13267. TRACKPOP was archived using the PKware file-compression utility.

Additional reference words: 3.10 softlib TRACKPOP.ZIP
KBCategory:
KBSubcategory: UsrMenTrackpop

INF: Switching Between Accelerator Tables in an Application

Article ID: Q80887

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In some cases, it is useful to change the accelerator keys that are assigned to menu items in an application. CHGACCL is a sample in the Software/Data Library that demonstrates changing the accelerators by loading a different accelerator table from the RC file and modifying the text of menu items.

CHGACCL can be found in the Software/Data Library by searching on the word CHGACCL, the Q number of this article, or S13283. CHGACCL was archived using the PKware file-compression utility.

DYNACCEL is a second sample in the Software/Data Library that demonstrates building an accelerator table as part of an application. For more information on DYNACCEL, query this knowledge base on the words:

   prod(winsdk) and dynaccel

More Information:

CHGACCL has two accelerator tables defined in the RC file and a menu item that allows the user to "swap" accelerator tables. When the user chooses to swap the accelerators, the application loads the second accelerator table and modifies the menu to reflect the changes in the accelerators. The user can swap back to the original accelerator table by choosing swap accelerators from the menu a second time.

Note that two (or more) instances of the application can be running, using different accelerators for each instance.

CHGACCL also uses specific GDI functions to properly draw shapes in the client area of the window. Each time the user chooses a new shape from the menu, the client area device context is reset to properly define the window's background color, pen color, mapping mode, and compression factors.

Additional reference words: 3.00 softlib CHGACCL.ZIP
KBCategory:
KBSubcategory: UsrMenRare/misc

Owner-Draw Menu Item Width Includes a Check Mark Width
Article ID: Q71061

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When an application creates an owner-draw menu item with the MF_OWNERDRAW style, the application receives a WM_MEASUREITEM message for that item. The application is required to fill the MEASUREITEMSTRUCT pointed to by the lParam of this message.

The itemWidth field of the MEASUREITEMSTRUCT is normally filled with the actual width of the item. However, when Windows displays that menu item, the width is increased by the width of a check mark; that is, Windows automatically expands the item to leave space for a check mark.

To make the menu item only as wide as the actual item, fill the itemWidth field with the width of the item, minus the width of a check mark, as returned by GetMenuCheckMarkDimensions().

The Software/Data Library contains an example, MENUBMP, of owner-draw menus with bitmaps where each item is only as large as the bitmap. MENUBMP can be found in the Software/Data Library by searching on the keyword MENUBMP, the Q number of this article, or S12971. MENUBMP was archived using the PKware file-compression utility.

More Information:

The following code fragment, built with the Windows Software Development Kit (SDK) version 3.00, demonstrates processing a WM_MEASUREITEM message that results in a menu item only as wide as the actual item. In this case, the menu is being made with bitmaps. The handle to the bitmap is stored as part of the item data.

```
        // Local variables.
        LPMEASUREITEMSTRUCT     lpItem;
        HBITMAP                 hBitmap;
        BITMAP                  bm;
        WORD                    cxCheck;

        ...

        case WM_MEASUREITEM:
            // lParam is a pointer to the structure.
            lpItem = (LPMEASUREITEMSTRUCT)lParam;

            // A bitmap handle was stored in the item data.
            hBitmap = LOWORD(lpItem->itemData);

            /*
```

```
             * The width of a check mark is automatically added to
             * menu items so we need to subtract it to make the
             * menu the minimum size.
             */
            cxCheck = LOWORD(GetMenuCheckMarkDimensions());

            // Get the bitmap dimensions
            GetObject(hBitmap, sizeof(BITMAP), (LPVOID)&bm);

            // Set the width to the width of the bitmap - cxCheck
            lpItem->itemWidth = bm.bmWidth - cxCheck;

            // Add one to the bitmap height for some spacing.
            lpItem->itemHeight = bm.bmHeight + 1;

            break;

        ...
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenPainting

INF: Sample Code Demonstrates Adding Menus Dynamically

Article ID: Q24600

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There are two different methods that can be used to add a pop-up menu to the menu bar at run time.

The first method is to define a version of the application's main menu that includes the pop-up. This is done in the resource (RC) file with any other menus that the application uses. To add the menu to the menu bar, use the following sequence of code:

```
hMenu = LoadMenu(hInst, (LPSTR)"<menu name>");
SetMenu(hWnd, hMenu);
DrawMenuBar(hWnd);
```

The second method creates the pop-up menu dynamically at run time. To use this method, place the following code in the application:

```
hPopup = CreateMenu();
AppendMenu(hPopup, MF_ENABLED, ID1, (LPSTR)"text 1");
AppendMenu(hPopup, MF_ENABLED, ID2, (LPSTR)"text 2");
hMenu = GetMenu(hWnd);
AppendMenu(hMenu, MF_POPUP, hPopup, (LPSTR)"new pop-up");
DrawMenuBar(hWnd);
```

There is a complete example of this dynamic-creation method in the Software Library as ADDMENU. ADDMENU can be found in the Software/Data Library by searching on the word ADDMENU, the Q number of this article, or S12070. ADDMENU was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenCreatemen

INF: Keeping Status Line Information About Menus Up-to-Date

Article ID: Q67689

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When a pop-up menu is opened with the mouse (by clicking on the menu), various items in the menu are highlighted as the mouse is moved. As the highlighted item changes, a WM_MENUSELECT message is sent to the application. This message can be processed to allow the application to update a status line.

If the mouse is moved from the pop-up menu back into the menu bar, no item in the pop-up menu is highlighted. However, no WM_MENUSELECT message is sent to the application. Instead, Windows sends the application a WM_ENTERIDLE message.

To keep the status line current, perform hit testing to determine if a pop-up menu item is selected during processing of the WM_ENTERIDLE message.

The Software Library contains sample code to demonstrate these procedures. MENUSTAT can be found in the Software/Data Library by searching on the word MENUSTAT, the Q number of this article, or S12848. MENUSTAT was archived using the PKware file-compression utility.

Additional reference words: 3.00 3.0
KBCategory:
KBSubcategory: UsrMenSelection

INF: Managing Per-Window Accelerator Tables

Article ID: Q82171

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Windows environment, an application can have several windows, each with its own accelerator table. This article describes a simple technique requiring very little code that an application can use to translate and dispatch accelerator key strokes to several windows. The technique employs two global variables, ghActiveWindow and ghActiveAccelTable, to track the currently active window and its accelerator table, respectively. These two variables, which are used in the TranslateAccelerator function in the application's main message loop, achieve the desired result.

More Information:

The key to implementing this technique is to know which window is currently active and which accelerator table, if any, is associated with the active window. To track this information, process the WM_ACTIVATE message that Windows sends each time an application gains or loses activation. When a window loses activation, set the two global variables to NULL to indicate that the window and its accelerator table are no longer active. When a window that has an accelerator table gains activation, set the global variables appropriately to indicate that the accelerator table is active. The following code illustrates how to process the WM_ACTIVATE message:

```
case WM_ACTIVATE:
   if (wParam == 0)  // indicates loss of activation
   {
      ghActiveWindow = ghActiveAccelTable = NULL;
   }
   else              // indicates gain of activation
   {
      ghActiveWindow = <this window>;
      ghActiveAccelTable = <this window's accelerator table>;
   }
   break;
```

The application's main message loop resembles the following:

```
while (GetMessage(&msg,      // message structure
                  NULL,      // handle of window receiving the msg
                  NULL,      // lowest message to examine
                  NULL))     // highest message to examine
{
   if (!TranslateAccelerator(ghActiveWindow, // active window
```

```
                             ghActiveAccelTable, // active accelerator
                             &msg))
   {
      TranslateMessage(&msg); // Translates virtual key codes
      DispatchMessage(&msg);  // Dispatches message to
                              //   window procedure
   }
}
```

Under Windows version 3.1, the WM_ACTIVATE message with the wParam set to WA_INACTIVE indicates loss of activation.

Additional reference words: 3.00 3.10 3.x SR# G920316-105
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Querying and Modifying the States of System Menu Items

Article ID: Q83453

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application should query or set states of the Restore, Move, Size, Minimize, and Maximize items on the system menu during the processing of a WM_INITMENU or a WM_INITMENUPOPUP message.

More Information:

Windows changes the state of the Restore, Move, Size, Minimize, and Maximize items on the system menu just before it draws the menu on the screen and sends the WM_INITMENU and WM_INITMENUPOPUP messages.

Windows sets the states of these menu items according to the state of the window just before the menu is displayed. For example, if the window is minimized when its system menu is pulled down, the Minimize menu item is unavailable (grayed). If an overlapped window is maximized when its system menu is pulled down, the Move, Size, and Maximize items are unavailable.

If an application queries or sets the state of any of these system menu items, the query or change should occur during the processing of the WM_INITMENU or WM_INITMENUPOPUP message. If any menu item state is queried before one of these messages is processed, it could reflect a previous state of the window. If any state is set before one of these messages is processed, Windows will reset the menu items to correspond to the state of the window just prior to sending these messages.

Windows does not change the state of the Close menu item. Its state can be changed or queried at any time.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMenSysmenu

INF:Win3.0 Top-Level Menu Items Unsupported in Owner-Draw Menu

Article ID: Q69969

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

Microsoft Windows version 3.0 allows an application to specify
owner-draw menus. This provides the application with complete control
over the appearance of items in the menu.

Windows 3.0 only supports owner-draw items in a pop-up menu.
Top-level menu items with the MF_OWNERDRAW style do not work properly.

In Windows 3.1, top-level menu items with the MF_OWNERDRAW style work
properly.

More Information:

An application may append an item with the MF_OWNERDRAW style to a
top-level menu. At this point, the parent application should receive a
WM_MEASUREITEM message and a WM_DRAWITEM message.

However, a WM_MEASUREITEM message is never sent to the parent window
for the menu item. In addition, the WM_DRAWITEM message is sent only
when the selection state of the item changes (the action field in the
DRAWITEMSTRUCT is equal to ODA_SELECTED). The WM_DRAWITEM message is
not sent with the action field in the DRAWITEMSTRUCT equal to
ODA_DRAWENTIRE.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMenPainting

PRB: SDK Sample Programs Define Delete Accelerator Incorrectly

Article ID: Q71147

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

SYMPTOMS

The Windows Software Development Kit (SDK) sample applications that define keyboard accelerators for the Edit menu define the Edit/Clear accelerator incorrectly.

RESOLUTION

To correct this problem, modify the RC file for each sample application to remove VK_SHIFT from the Edit/Clear accelerator.

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrMenCreatemen

INF: Various Ways to Access Submenus and Menu Items

Article ID: Q71454

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In calls to Microsoft Windows functions that create, modify, and destroy menus, an application can access an individual menu item by either its position or its item ID. A pop-up menu must be accessed by its position because it does not have a menu-item ID.

Specifically, when an application calls the EnableMenuItem function to enable, disable, or dim (gray) an individual menu item, the application can specify either the MF_BYPOSITION or the MF_BYCOMMAND flag in the wEnable parameter. When the application calls EnableMenuItem to access a pop-up menu, it must specify the MF_BYPOSITION flag.

The information below provides examples of the following:

 - Retrieving a menu handle for a submenu
 - Accessing a submenu
 - Accessing a menu item

More Information:

The following resource-file menu template provides the basis for the source code examples in this article. The template describes a top-level menu with two pop-up submenus. One of the submenus contains a third, nested submenu.

```
   GenericMenu MENU
   BEGIN
      POPUP "&Help"
      BEGIN
         MENUITEM "&About Generic...", IDM_ABOUT
      END

      POPUP "&Test"
      BEGIN
         POPUP "&Nested"
         BEGIN
            MENUITEM "&1 Beep", IDM_1BEEP
            MENUITEM "&2 Beeps", IDM_2BEEPS
         END
      END
   END
```

Retrieving the Handle to a Submenu

Code such as the following can be used to obtain handles to the menus:

```
HMENU hMainMenu, hHelpPopup, hTestPopup, hNestedPopup;

<other program lines>

hMainMenu = GetMenu(hWnd);
hHelpPopup = GetSubMenu(hMainMenu, 0);
hTestPopup = GetSubMenu(hMainMenu, 1);
hNestedPopup = GetSubMenu(hTestPopup, 0);
```

The second parameter of the GetSubMenu function, nPos, is the position of the desired submenu. Positions are numbered starting at zero for the first menu item.

Disabling a Submenu

The following call disables and dims the Nested pop-up menu:

```
EnableMenuItem(hTestPopup, 0, MF_BYPOSITION | MF_GRAYED);
```

The following call disables and dims the Test pop-up menu:

```
EnableMenuItem(hMainMenu, 1, MF_BYPOSITION | MF_GRAYED);
```

The second parameter of the EnableMenuItem function, wIDEnabledItem, is the position of the submenu. As above, positions are numbered starting at zero. Note that the call must specify the MF_BYPOSITION flag because a pop-up menu does not have a menu-item ID.

Disabling a Menu Item

The 1 Beep menu item can be disabled and dimmed by using any one of the following calls:

```
EnableMenuItem(hMainMenu, IDM_1BEEP, MF_BYCOMMAND | MF_GRAYED);
EnableMenuItem(hTestPopup, IDM_1BEEP, MF_BYCOMMAND | MF_GRAYED);
EnableMenuItem(hNestedPopup, IDM_1BEEP, MF_BYCOMMAND | MF_GRAYED);
EnableMenuItem(hNestedPopup, 0, MF_BYPOSITION | MF_GRAYED);
```

A menu item can be specified by either by its menu-item ID value (using the MF_BYCOMMAND flag) or by its position (using the MF_BYPOSITION) flag. If the application specifies the menu-item ID value, Windows must walk the menu structure and search for a menu item with the correct ID. This implies the each menu-item ID value must be unique for a given menu.

Other Windows Menu Functions

Although the EnableMenuItem function is used in the example above, the

same general approach is used for all Windows menu functions; access pop-up menus by position, and access menu items by position or menu-item ID.

For a list of all Windows menu functions, see page 1-56 of the "Microsoft Windows Software Development Kit Reference--Volume 1." For more information on working with menus in a Windows application, see Chapter 7 of the "Microsoft Windows Software Development Kit Guide to Programming" for version 3.0.

Additional reference words: 3.00 dimmed unavailable
KBCategory:
KBSubcategory: UsrMen

INF: Appending Menu Items to Other Applications
Article ID: Q72222

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To enhance the functionality of an existing Windows application, new menu items can be added to the target application. The source application that adds the new menu items can also respond to these menu items when they are selected by a user. This article discusses the development of such a source application and the pitfalls associated with using this type of application.

There is sample code in the Software/Data Library, in a file called SUBAPP, that demonstrates the feature of adding menu items to another Windows application.

SUBAPP can be found in the Software/Data Library by searching on the word SUBAPP, the Q number of this article, or S13099. SUBAPP was archived using the PKware file-compression utility.

More Information:

To monitor the menu items of another application, the main window of that application is subclassed using the SetWindowLong() function. In the new window function, WM_SYSCOMMAND messages should be processed to handle the system menu items. If the menu items belong to a pop-up menu, WM_COMMAND should be processed. All the other messages and unrelated menu messages should be passed to the previous window function to maintain the integrity of the target application.

After subclassing the target application, new menu items or new pop-up menus can be added to it. To add menu items to the system menu or an existing pop-up menu, AppendMenu() is called, and to add new pop-up menus, InsertMenu() is called.

To make this process work in all three modes of Windows, a fixed code dynamic-link library (DLL) should be used. The DLL should contain the function that adds menu items and the new window function for the target application. This is because the code associated with such a mechanism must be available at all times. Certain EMS (expanded memory specification) memory configurations place all code except fixed library code segments in application-specific EMS memory. This makes the code availability limited to the application that owns the EMS memory. If that code is not available to the target application, it will crash under those EMS configurations.

Note: For future compatibility, the subclass procedure must be in a DLL for all modes.

Care should be taken when assigning ID (identification) values to the new menu items. The new ID values should not conflict with the values of the existing menu items; otherwise, the old menu items will be disabled. To determine the existing ID values of the target application, use SPY (included with the Windows Software Development Kit version 3.00) to monitor WM_MENUSELECT messages when selecting menu items. The wParam parameter of WM_MENUSELECT message contains the ID value of the menu item selected.

Some applications, such as Microsoft Excel and Microsoft Word for Windows, use more than one menu; that is, each application has an option to use a long menu or a short one. These applications also allow their users to customize the menus. These features could become a problem when subclassing this type of application and monitoring new pop-up menus or new menu items.

For example, when subclassing Word for Windows, a new menu item is added under the pop-up menu "Utility". Suppose that, when appending the new item, the long version of the menu was selected. Later on, if the user changes to the short version of the menu, the new added menu item will be lost because Word for Windows would load the short version of its menus. To avoid this problem, add menu items to the system menu of Word for Windows -- changing the menu version does not effect the system menu.

Microsoft Excel and Microsoft Word for Windows can also display the menu items selected in a status window at the bottom. For this purpose, these applications use the WM_MENUSELECT message. If these applications receive a WM_MENUSELECT for a menu item that was not originally a part of their menu, they could crash when they refresh the status window. For this reason, in the new window function, WM_MENUSELECT messages related to all the new menu items should not be passed to the target application's old window function.

If there are a lot of new menu items to be added to an application, cascading menus should be used. Such a menu structure can help minimize the number of commands on an original pop-up menu of the target application.

As previously stated, there is a sample in the Software/Data Library called SUBAPP that demonstrates the feature of adding menu items to another Windows application. In WinMain of SUBAPP.EXE, FindWindow() is called to get the handle of the desired application. If the application is present in the system, its main-window is subclassed with a new function and two new menu items are appended to its system menu. The new function monitors the messages of the target application. If any of those messages relate to the newly added menu items, the messages are handled accordingly; otherwise, they are sent to the target application's original function.

If, on the other hand, the application is not present, a message hook is installed to monitor the execution of the desired application. The message hook waits for the WM_PAINT message to arrive for the desired window class.

Upon receiving a WM_PAINT for the desired application, the callback hook function subclasses the other application and appends menu items

to its system menu. The hook function then posts a message to the calling application (SUBAPP.EXE) to unhook the message filter, since there is no longer a need for a system message hook.

If the other application quits before the termination of SUBAPP.EXE, a message hook is set that again waits for the activation of the desired application to append the menu items. If SUBAPP.EXE is closed before the target application, the added menu items are removed and the application is no longer subclassed.

The message hook, the subclassing function for the application, and all of the menu appending and removing calls are placed in a fixed code DLL, called LIBSUBAP.DLL. This guarantees that SUBAPP.EXE will work properly in all of the three modes of Windows.

SUBAPP.EXE can be used to subclass any Windows application as long as the class name of the target application's main window is known. In the application's header file (SUBAPP.H), the constant OTHER_CLASS_NAME can be initialized to the desired application's main-window class name.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Initializing Menus Dynamically

Article ID: Q75630

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Many commercial applications developed for Windows allow the user to customize the menus of the application. This ability introduces some complexity when the application must disable particular menu items at certain times. This article provides a method to perform this task.

More Information:

Windows sends a WM_INITMENUPOPUP message just before a pop-up menu is displayed. The parameters to this message provide the handle to the menu and the index of the pop-up menu on the main menu.

To process this message properly, each menu item must have a unique identifier. When the application starts up, it creates a mapping array that lists the items on each menu. When the WM_INITMENUPOPUP message is received, the application checks the conditions necessary for each menu item to be disabled or checked and modifies the menu appropriately.

The application must maintain the mapping array when the user modifies the menus in any way.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenCreatemen

PRB: FatalExit 0x0504 Produced from Incorrect lpTableName

Article ID: Q41453

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

When calling the LoadAccelerators() function, FatalExit code 0x0504 occurs.

CAUSE

The accelerator table named by the lpTableName parameter is incorrect.

RESOLUTION

The lpTableName parameter must match an accelerator table name in the application's resources.

Additional reference words: 1.x 2.03 2.10 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: GetMenuState() Can Return MF_BITMAP
Article ID: Q74700

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A description of the MF_BITMAP return value for the GetMenuState() function was omitted from pages 4-186 of the "Microsoft Windows Software Development Kit Reference Volume 1."

If the menu item specified in the call to GetMenuState is a bitmap, GetMenuState will set the MF_BITMAP bit in its return value. Although MF_BITMAP is not included in the GetMenuState documentation, it is defined in the WINDOWS.H header file, and is fully supported.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

PRB: One Cause of RIP 0x0140 in USER When Accessing Menu

Article ID: Q74736

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

When running an application under the debugging version of Windows 3.0, USER fatal exits with error code 0x0140 when accessing a menu.

CAUSE

The TranslateAccelerator function is not re-entrant in Windows version 3.0 and, if re-entered, is likely to cause a FatalExit (RIP) 0x0140 when a menu is accessed.

Applications may use a separate message loop to perform background processing for a process, such as repaginating a large document or recalculating a large table. Often this process is started in response to a particular WM_COMMAND message that may be generated by a menu selection or a keyboard accelerator.

Inside this secondary message loop, the application may again call TranslateAccelerator. When this process terminates and the original WM_COMMAND message is completely processed, the menu will have been destroyed.

RESOLUTION

To avoid this situation, an application should post a message to itself to begin the processing. This allows TranslateAccelerator to finish before the processing begins and the secondary message loop is entered.

More Information:

When TranslateAccelerator detects a keystroke that is present in the accelerator table, it sets a flag in the internal menu structure and sends a WM_COMMAND message to the application. The flag set in the structure indicates that TranslateAccelerator is in a SendMessage state.

When this SendMessage call returns, TranslateAccelerator clears this flag if it is set, or destroys the menu if it is clear. A menu must only be destroyed when an accelerator caused a menu to be displayed. This does not occur when an application is running in a normal or maximized state.

If TranslateAccelerator is called a second time from within the processing of the first WM_COMMAND message, it will clear the flag set in the menu structure. When the first call to TranslateAccelerator returns from its SendMessage, the cleared flag causes the application's top-level menu to be destroyed. The next time the menu

is accessed, USER will RIP with code 0x0140: Invalid Local Handle because the menu handle was invalidated with DestroyMenu.

By posting a message in response to the first WM_COMMAND message, the application allows TranslateAccelerator to return from SendMessage and exit, which ensures the menu state is properly maintained.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

PRB: TrackPopupMenu() on LoadMenuIndirect() Menu Causes UAE

Article ID: Q75254

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

When LoadMenuIndirect() is used to create a menu from a menu template and the menu handle is passed to TrackPopupMenu(), Windows reports an unrecoverable application error (UAE).

CAUSE

The menu handle returned from LoadMenuIndirect() does not point to a menu with the MF_POPUP bit set.

RESOLUTION

The following code fragment demonstrates the correct procedure to "wrap" the menu created by LoadMenuIndirect() inside another menu. This procedure sets the MF_POPUP bit properly.

```
hMenu1 = LoadMenuIndirect(lpMenuTemplate);

hMenuDummy = CreateMenu();
InsertMenu(hMenuDummy, 0, MF_POPUP, hMenu1, NULL);

hMenuToUse = GetSubMenu(hMenuDummy, 0);
```

Use hMenuToUse when TrackPopupMenu() is called. The values of hMenu1 and hMenuToUse should be the same.

When the menu is no longer required, call DestroyMenu() to remove hMenuDummy. This call will also destroy hMenu1 and free the resources it used.

Additional reference words: 3.00 3.0
KBCategory:
KBSubcategory: UsrMenTrackpop

Reasons for Failure of Menu Functions
Article ID: Q92409

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The Menu functions (AppendMenu, CheckMenuItem, CreateMenu, CreatePopupMenu, DeleteMenu, DestroyMenu, GetMenu, GetMenuItemID, GetMenuString, GetSubMenu, GetSystemMenu, HiliteMenuItem, InsertMenu, LoadMenuIndirect, ModifyMenu, RemoveMenu, SetMenu, SetMenuItemBitmaps, and TrackPopupMenu) can fail for several reasons. Different functions return different values to indicate failure. Read the documentation for information about each function. This article combines the causes of failure for all functions and provides a resolution or explanation. A list of affected functions follows each cause in the More Information section. The causes are:

1. Invalid hWnd parameter.

2. Invalid hMenu parameter.

3. The menu item is not found.

4. No space left in User's heap to hold a string or to hold an internal data structure for owner draw menu items or to create a menu or to create a window for TrackPopupMenu.

5. There are no items in the menu.

6. The menu resource could not be found (FindResource) or loaded (LoadResource) or locked (LockResource) in memory.

7. TrackPopupMenu is called while another pop-up menu is being tracked in the system.

8. The hMenu that has been passed to TrackPopupMenu has been deleted.

9. MENUITEMTEMPLATEHEADER 's versionNumber field is non-zero.

More Information:

Cause 1: Invalid hWnd parameter.

Resolution 1: Validate the hWnd parameter using IsWindow. Make sure that hWnd is not a child window.

Explanation 1: In Windows, menus are always associated with a window. Child windows cannot have menus.

Affected Functions: All functions that take hWnd as a parameter.

Cause 2: Invalid hMenu parameter.

Resolution 2: Validate hMenu with IsMenu.

Affected Functions: All functions that take hMenu as a parameter.

Cause 3: The menu item is not found.

Resolution 3: If the menu item is referred to BY_POSITION, make sure that the index is less than the number of items. If the menu item is referred to BY_COMMAND, an application must devise its own method of validating it.

Explanation 3: Menu items are numbered consecutively starting from 0. Remember that separator items are also counted.

Affected Functions: All functions that refer to a menu item.

Cause 4: No space left in User's heap to hold a string or to hold an internal data structure for owner-draw menu items or to create a menu.

Resolution 4: Remember, when they are not needed any more, delete all menus and other objects that have been created by the application. If you suspect that objects left undeleted by other applications are wasting valuable system resources, restart Windows.

Explanation 4: In Windows 3.0, menus and menu items are allocated space from User's heap. In Windows 3.1, they are allocated space from a separate heap. This heap is for the exclusive use of menus and menu items.

Affected Functions: AppendMenu, Insertmenu, ModifyMenu, CreateMenu, CreatePopupMenu, LoadMenu, LoadMenuIndirect, TrackPopupMenu, GetSystemMenu (when fRevert = FALSE).

Cause 5: There are no items in the menu.

Resolution 5: Use GetMenuItemCount to make sure the menu is not empty.

Explanation 5: Nothing to be deleted or removed.

Affected Functions : RemoveMenu, DeleteMenu.

Cause 6: The menu resource could not be found (FindResource) or loaded (LoadResource) or locked (LockResource) in memory.

Resolution 6: Ensure that the menu resource exists and that the hInst parameter refers to the correct hInstance. Try increasing the number of file handles using SetHandleCount and increasing available global memory by closing some applications. For more information about the causes of failure of resource functions, query this knowledge base on the following words:

    prod(winsdk) and failure and LoadResource and FindResource and

LockResource.
Explanation 6: Finding, loading, and locking a resource involves use of file handles, global memory, and the hInstance that has the menu resource.

Affected Functions: LoadMenu, LoadMenuIndirect

Cause 7. TrackPopupMenu is called while another pop-up menu is being tracked in the system.

Explanation 7: Only one pop-up menu can be tracked in the system at any given time.

Affected Function: TrackPopupMenu

Cause 8. The hMenu that was passed to TrackPopupMenu was deleted. The debug mode of Windows 3.1 sends the following message :

Menu destroyed unexpectedly by WM_INITMENUPOPUP

Explanation 8: Windows sends a WM_INITMENUPOPUP to the application and expects the menu to not be destroyed.

Affected Function: TrackPopupMenu

Cause 9. MENUITEMTEMPLATEHEADER 's versionNumber field is non-zero.

Explanation 9: In Windows 3.0 and 3.1, this field should always be 0.

Affected Function: LoadMenuIndirect

Additional reference words: 3.00 3.10 fails
KBCategory:
KBSubcategory: UsrMen

INF: Creating Accelerator Tables Dynamically

Article ID: Q75738

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Sometimes it is desirable to allow the user to define accelerator keys when running an application. No Windows function provides this functionality, nor does the typical method of using the Resource Compiler (RC) and embedding the accelerator table in the .EXE file. To accomplish this, the programmer must create and manage an accelerator table.

This article outlines one method of creating accelerator tables at run time or "on the fly." The DYNACCEL sample, which is available in the Software/Data Library, demonstrates this method.

More Information:

DYNACCEL demonstrates creating an accelerator table at run time rather than in the resource file. It creates a table of accelerator keys and their associated command identifiers. It implements a CustomTranslateAccelerator function, which is used in place of Windows' TranslateAccelerator function. It traps any WM_KEYDOWN messages, checks the table to see if the key matches an accelerator key, checks to see if the CTRL or SHIFT key is down as appropriate, and generates a WM_COMMAND message if all conditions are met.

DYNACCEL can be enhanced to use a dynamically sized or updated table or to include other desired functionality because everything is created and maintained within the program.

The following code fragments are used in DYNACCEL to implement an accelerator table at run time. The complete DYNACCEL sample can be found in the Software/Data Library by searching on the word DYNACCEL, the Q number of this article, or S13147. DYNACCEL was archived using the PKware file-compression utility.

```
/*
 * Accelerator table data structure. This has been set up arbitrarily
 * and may be altered depending on the desired functionality.
 */

typedef struct {
    BYTE cChar;
    BYTE cVirtKey;
    WORD wID;
} Accelerator;
```

```
/*
 * Array of accelerator keys. For demonstration purposes, these values
 * are predefined by DYNACCEL. An application could read in the
 * user's keystrokes and assign these values at run time or from an
 * INI file.
 */

Accelerator AccTable[TABLELENGTH] =   {
                {'A', VKCTRL, IDDA},        //CTRL+key will activate
                {'B', VKCTRL, IDDB},        //    "     "      "
                {'C', VKCTRL, IDDC},        //    "     "      "
                {'D', VKSHFT, IDDD},        //SHIFT+key will activate
                {'E', VKSHFT, IDDE},        //    "     "      "
                {'F', VKSHFT, IDDF},        //    "     "      "
                {' ', 0, IDD_UNDEFINED},    //not defined
                {' ', 0, IDD_UNDEFINED}
                                      };


/*
 * FUNCTION: CustomTranslateAccelerator(HWND, LPMSG)
 *
 * PURPOSE: Custom TranslateAccelerator for dynamic accelerator
 *          tables.
 */

int CustomTranslateAccelerator(HWND hWnd,
                               LPMSG lpMsg)
{
int i;

if (lpMsg->message == WM_KEYDOWN)
   for (i = 0; i < TABLELENGTH; i++)  //check table for a match
      {
      if ((lpMsg->wParam == AccTable[i].cChar)   //key is in table
           && (AccTable[i].wID != IDD_UNDEFINED))  //and is defined

         switch (AccTable[i].cVirtKey)
            {
            case VKCTRL:  //is CONTROL down?
               if (GetKeyState(VK_CONTROL) & 0x1000)
                  return ActivateMenuItem(hWnd, AccTable[i].wID);
               break;

            case VKSHFT:  //is SHIFT down?
               if (GetKeyState(VK_SHIFT) & 0x1000)
                  return ActivateMenuItem(hWnd, AccTable[i].wID);
               break;
            } //end switch
      } //end for loop

   return FALSE;    //key combination not found in accelerator table
}


/*
 * FUNCTION: ActivateMenuItem(HWND, WORD)
```

```
 *
 * PURPOSE: Highlight (HILITE) a top level menu item, run the command
 *          associated with a menu item, and unhighlight (UNHILITE)
 *          the top-level menu item.
 */

BOOL ActivateMenuItem(HWND hWnd,
                      WORD wID)
{
HiliteMenuItem(hWnd, GetMenu(hWnd), wID, MF_BYCOMMAND | MF_HILITE);
SendMessage(hWnd, WM_COMMAND, wID, 0L);
HiliteMenuItem(hWnd, GetMenu(hWnd), wID, MF_BYCOMMAND | MF_UNHILITE);
return TRUE;  //key combination found and processed
}


/*
 * Implementation of CustomTranslateAccelerator within WinMain
 */

    while (GetMessage(&msg, NULL, NULL, NULL))
        {
        if (CustomTranslateAccelerator(hwnd, (LPMSG)&msg) == 0)
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }
```

Additional reference words: 3.00 3.0
KBCategory:
KBSubcategory: UsrMenDynaccel

INF:Accessing Parent Window's Menu from Child Window w/ focus

Article ID: Q92527

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

In an MDI-like application, the user must be allowed to pull down menus in the parent window by using menu mnemonics even though the child window or one of its children may have the focus. This can be done by creating child windows without a system menu or by processing the WM_MENUCHAR and WM_SYSCOMMAND/SC_KEYMENU messages to programatically pull down the parent's menu.

More Information:

If a child window with a system menu has the focus and the user attempts to access the parent's menu with the keyboard using the menu mnemonic (ALT+mnemonic character), Windows will beep and the parent's menu will not be pulled down. This problem occurs because the parent window does not have the focus and because the window with the focus does not have a menu corresponding to the mnemonic. (Child windows cannot have menus other than the system menu.)

If the child window with the focus does not have a system menu, Windows assumes that the menu mnemonic is for the nearest ancestor with a system menu and passes the message to that parent. Consequently, it is possible to use menu mnemonics to pull down a parent's menu if the descendant windows do not have system menus.

If the child window with the focus has a system menu, Windows will beep if a menu mnemonic corresponding to a parent menu is typed. This can be prevented and the parent menu can be dropped down using the following code in the window procedure of the child window:

```
 case WM_MENUCHAR:
    PostMessage(hwndWindowWithMenu, WM_SYSCOMMAND, SC_KEYMENU, wParam);
    return(MAKELRESULT(0, 1));
```

WM_MENUCHAR is sent to the child window when the user presses a key sequence that does not match any of the predefined mnemonics in the current menu. wParam contains the mnemonic character. The child window posts a WM_SYSCOMMAND/SC_KEYMENU message to the parent whose menu is to be dropped down, with lParam set to the character that corresponds to the menu mnemonic.

The above code can also be used if the child window with the focus does not have a system menu but an intermediate child window with a system menu exists between the child with the focus and the ancestor whose menu is to be dropped. In this case, the code would be placed in the intermediate

window's window procedure.

Additional reference words: 3.10 3.00
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Control Accelerators Conflict With Their ANSI Equivalents

Article ID: Q59388

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

If the CTRL+H and CTRL+M key combinations are defined as accelerators, Windows translates them as the BACKSPACE and ENTER (RETURN, on some keyboards) keys, respectively. This causes child edit controls and those in modeless dialog boxes to behave incorrectly.

More Information:

This improper translation occurs with windows that process messages within the same message loop as the window that owns the accelerators. These windows are usually child windows and modeless dialog boxes. The following table lists some common keys that cause conflicts with edit controls when used as accelerators:

```
        Accelerator          ANSI Definition
        -----------          ---------------

        CTRL+H               BACKSPACE (VK_BACK   0x08)
        CTRL+I               HTAB      (VK_TAB    0x09)
        CTRL+M               ENTER     (VK_RETURN 0x0D)
```

These keystrokes are processed during the WM_KEYDOWN message. For example, when CTRL+H is pressed, a WM_KEYDOWN is processed for the CTRL key (VK_CONTROL 0x11). If another WM_KEYDOWN is processed for the letter "H" (0x48), TranslateAccelerator() posts a WM_COMMAND message to the owner of the CTRL+H accelerator. When the BACKSPACE key is pressed, a WM_KEYDOWN is also processed with VK_BACK (0x08) as the keycode. Apparently, Windows processes this key as its CTRL+H ANSI equivalent. The two key sequences cause a WM_COMMAND to be sent to the owner of the CTRL+H accelerator, thus causing the child window with the input focus to miss the BACKSPACE key.

The opposite does not apply. When the BACKSPACE, TAB, and ENTER keys are used as accelerators, Windows does not map them onto the matching control sequences listed above.

To work around this feature, your application can process these control sequences manually and not use them as accelerators, or the window that is affected can be subclassed and the desired key sequence(s) trapped before the owner gets a WM_COMMAND from TranslateAccelerator(). The following is one way to do this:

```
/*
     The code below subclasses a child edit box to allow it to process
     the ENTER and BACKSPACE key while allowing the owner to receive
     WM_COMMAND messages for its CTRL+H and CTRL+M accelerators.
```

```
 */

...
long FAR PASCAL NewEditProc( HWND, unsigned, WORD, LONG );

FARPROC lpfnOldEditProc;
HWND    hWndOwner;

...
long FAR PASCAL NewEditProc( HWND hWnd, unsigned iMessage, WORD wParam,
                                            LONG lParam )
{
    MSG     msg;

switch ( iMessage )
{
    case WM_KEYDOWN:
         switch ( wParam )
         {
             case VK_BACK:
               // This assumes that the next message in the queue will be
               //  a WM_COMMAND for the OWNER.
                  PeekMessage( &msg, hWndOwner, 0, 0, PM_REMOVE );

               // Since TranslateAccelerator() processed this message as
               //  an accelerator, WM_CHAR must be supplied manually
               //  because it is not posted to this window.
                  SendMessage( hWnd, WM_CHAR, wParam, MAKELONG( 1, 14 ) );
                  return 0L;

             case VK_RETURN:
               // Same procedures here.
                  PeekMessage( &msg, hWndOwner, 0, 0, PM_REMOVE );
                  SendMessage( hWnd, WM_CHAR, wParam, MAKELONG( 1, 28 ) );
                  return 0L;
         }
         break;
    }
    return CallWindowProc( lpfnOldEditProc, hWnd, iMessage, wParam, lParam );
}
```

Note that hWndOnwer is the window that owns the accelerators. If this edit box was in a modeless dialog box, hWndOwner should be NULL to work properly. It can also be NULL in the above case.

Additional reference words: 2.03 2.10
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Tracking Menu Selections in Windows Programs
Article ID: Q76892

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article discusses how to provide feedback to users of a program created with the Microsoft Windows Software Development Kit (SDK) version 3.0, as different menu items are selected.

More Information:

When a user selects an item from a menu, an application can provide a simple explanation of the menu item by updating a text string in a status area of the main window.

This can be done easily because the main window procedure will receive a WM_MENUSELECT message for each item chosen. When this message is received, wParam is the integer ID of the menu item (as defined in the RC file), LOWORD(lParam) contains selection flags, and HIWORD(lParam) is a handle to the selected menu item. By indexing on the value in wParam, the appropriate string can be loaded from a string table and displayed.

However, if the item that the user selects is a pop-up menu, there won't be an associated integer ID for that item. In this case, wParam is not useful to determine which item is selected. Instead, the application must also process the WM_INITMENUPOPUP message. When this message is received by the program, wParam is the pop-up menu handle, LOWORD(lParam) is the index of the pop-up menu, and HIWORD(lParam) is either 1 (the system menu) or 0 (any other menu). By checking LOWORD(lParam), the application can then load the appropriate string from the string table and display it in the message area.

The MENUSTR.ZIP file in the Software/Data Library demonstrates these methods. MENUSTR can be found in the Software/Data Library by searching on the word MENUSTR, the Q number of this article, or S13195. MENUSTR was archived using the PKware file-compression utility.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMenSelection

INF: Sample Code Implementing a Child Window with Menus
Article ID: Q93199

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Child windows by design do not have menus; however, some applications may require such child windows. CHILDMNU demonstrates one way to embed a pop-up window with menus in a child window to simulate a child window with menus.

CHILDMNU can be found in Software/Data Library by searching on the word CHILDMNU, the Q number of this article, or S14125. CHILDMNU was archived using the PKware file-compression utility.

More Information:

Child windows were not designed to have menus but under certain circumstances an application may require a child window with menus. Below are the steps to simulate a child window with menus:

1. Create the child for a main window using standard methods.

2. Create a pop-up window without a caption or border. The pop-up window should take up the entire client area of the child window.

3. Moving: Set the child window as the parent of the pop-up window using SetParent(). This method clips the pop-up window to the child's client and moves the pop-up window relative to the child window when the child moves.

4. Sizing: If the child window has sizing borders, then the pop-up window must be adjusted as the child is sized.

   Correct sizing of the pop-up window can be accomplished by processing the child window's WM_SIZE message. When the child window receives a WM_SIZE message, the child's new client area is calculated and the pop-up window is adjusted.

5. Activation: Child windows are not activated by design. When the user clicks the pop-up window, activation is taken away from the main window and is given to the pop-up window. To simulate the main window's activation, the pop-up's window procedure, upon receiving a WM_ACTIVATE message, posts a WM_NCACTIVATE message to the main window.

   Note: This step causes flashing because the caption is repainted when going back and forth between the pop-up window and the main window. Unfortunately, there is no sure way to accomplish the change of caption's activation.

Other Considerations

Maximizing: When a child window is maximized, it uses the screen's maximum size. To have the child window act similar to an MDI child, two messages need to be processed; WM_SIZE in the main parent's window procedure and WM_MINMAXINFO in the child's window procedure.

WM_SIZE of the child's parent window procedure checks for SIZE_MAXIMIZE and SIZE_RESTORE. Then, if the child window is maximized, call ShowWindow() with SW_MAXIMIZED. This method adjusts the child window's size to the new size of the parent's client area.

On WM_MINMAXINFO of the child's window procedure, calculate the size of the parent's client area. Use the newly calculated size and set the ptMaxSize variable of the MINMAXINFO structure. This ensures that the child is always fully contained within its parent client area when maximized.

Additional reference words: 3.10 softlib CHILDMNU.ZIP
KBCategory:
KBSubcategory: UsrMenRare/misc

INF: Mirroring Main Menu with TrackPopupMenu()

Article ID: Q99806

Summary:

A developer may want to use TrackPopupMenu() to display the same menu that is used by the main window. TrackPopupMenu() takes a pop-up menu, while GetMenu() and LoadMenu() both return handles to top level menus, and therefore you cannot use the menus returned from these functions with TrackPopupMenu(). To "mirror" the main menu, you must create pop-up menus with the same strings, style, and IDs as the main menu. To do this, use the following Windows APIs:

```
GetMenu()
CreatePopupMenu()
GetMenuState()
GetMenuString()
GetSubMenu()
AppendMenu()
```


More Information:

The following code displays the same menu as the main window when the right mouse button is clicked:

```
// In the main window procedure...

    case WM_RBUTTONDOWN:
    {

           HMENU hMenu;             // The handle to the main menu.
           int nMenu;               // The index of the menu item.
           POINT pt;              // The point to display the track menu.
           HMENU hMenuOurs;         // The pop-up menu that we are creating.
           UINT nID;              // The ID of the menu.
           UINT uMenuState;         // The menu state.
           HMENU hSubMenu;         // A submenu.
           char szBuf[128];         // A buffer to store the menu string.

// Get the main menu.
           hMenu = GetMenu(hWnd);
           nMenu = 0;

// Create a pop-up menu.
           hMenuOurs = CreatePopupMenu();

// Get menu state will return the style of the menu
// in the lobyte of the unsigned int. Return value
// of -1 indicates the menu does not exist, and we
// have finished creating our pop up.
           while ((uMenuState =
              GetMenuState(hMenu,nMenu,MF_BYPOSITION)) != -1)
           {
                if (uMenuState != -1)
                {
// Get the menu string.
```

```
                GetMenuString(hMenu,nMenu, szBuf,128,MF_BYPOSITION);
                      if (LOBYTE(uMenuState) & MF_POPUP) // It's a pop-up menu.
                      {
                          hSubMenu = GetSubMenu(hMenu,nMenu);
                             AppendMenu(hMenuOurs,
                      LOBYTE(uMenuState),hSubMenu,szBuf);
                      }
                      else  // Is a menu item, get the ID.
                      {
                          nID = GetMenuItemID(hMenu,nMenu);
                          AppendMenu(hMenuOurs,LOBYTE(uMenuState),nID,szBuf);

                      }
                      nMenu++;  // Get the next item.
                  }
        }
        pt = MAKEPOINT(lParam);
// TrackPopupMenu expects screen coordinates.
        ClientToScreen(hWnd,&pt);
           TrackPopupMenu(hMenuOurs,
          TPM_LEFTALIGN|TPM_RIGHTBUTTON,
          pt.x,pt.y,0,hWnd,NULL);

// Because we are using parts of the main menu in our
// pop-up menu, we can't just delete the pop-up menu, because
// that would also delete the main menu. So we must
// go through the pop-up menu and remove all the items.
        while (RemoveMenu(hMenuOurs,0,MF_BYPOSITION))
            ;

// Destroy the pop-up menu.
        DestroyMenu(hMenuOurs);
    }
    break;
```

If the menu is never dynamically modified, then the menu hMenuOurs could be made static and created inside the WM_CREATE message, and destroyed in the WM_DESTROY message.

To see how this function works, paste this code into the MENU sample application shipped with both Microsoft Visual C/C++ and Microsoft C/C++ version 7.0 in the file MENU.C in the MenuWndProc() function.

Additional reference words: 3.10 popup
KBCategory:
KBSubcategory: UsrMenTrackpop

INF: Using SendMessage() As Opposed to SendDlgItemMessage()

Article ID: Q12273

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following information describes under what circumstances it is appropriate to use either the SendMessage() or SendDlgItemMessage() function.

Both SendMessage() and SendDlgItemMessage() can be used to add strings to a list box. SendMessage() is used to send a message to a specific window using the handle to the list box. SendDlgItemMessage() is used to send a message to the child window of a given window using the list box resource ID. SendDlgItemMessage() is most often used in dialog box functions that have a handle to the dialog box and not to the child window control.

The SendDlgItemMessage() call

```
SendDlgItemMessage(hwnd,id,msg,wParam,lParam)
```

is equivalent to the following SendMessage() call:

```
hwnd2 = GetDlgItem(hwnd,id);
SendMessage(hwnd2,msg,wParam,lParam);
```

Please note that PostMessage() should never be used to talk to the child windows of dialog boxes for the following reasons:

1. PostMessage() will only return an error if the message was not posted to the control's message queue. Since many messages are sent to control return information, PostMessage() will not work, since it does not return the information to the caller.

2. Messages such as the WM_SETTEXT message that include a far pointer to a string can potentially cause problems if posted using the PostMessage() function. The far pointer may point into a buffer that is inside the DS (data segment). Because PostMessage() does not process the message immediately, the DS might get moved. If the DS is moved before the message is processed, the far pointer to the buffer will be invalid.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMsgPostmsg

INF: Introduction to Nonpreemptive Multitasking in Windows

Article ID: Q78155

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Applications in the Windows environment are scheduled using a nonpreemptive multitasking system. Usually, multitasking operating systems schedule tasks for execution based on some combination of time slice and/or task priority. However, Windows does not preemptively switch the processor away from a currently running application under normal circumstances (excluding interrupts). Instead, multitasking in Windows is based on the messaging system.

Windows applications must cooperatively offer to yield the processor on a regular basis to allow Windows to multitask. Normally, an application offers to yield by calling one of three functions: GetMessage, PeekMessage, or WaitMessage. An application may also yield when it calls DialogBox, DialogBoxIndirect, DialogBoxParam, DialogBoxIndirectParam, MessageBox, or TrackPopupMenu.

More Information:

The GetMessage Loop

In the Windows environment, most applications have a central GetMessage loop that allows the application to cooperate with the other applications running in the system.

When an application (called "App A") calls GetMessage, the GetMessage function will return if there are any messages in App A's message queue. If there are no messages, Windows will give control to another application that has messages in its queue. If there are no messages waiting for any application in the system, Windows idles. App A's call to GetMessage will not return until there is a new message in App A's queue.

Using this method, the Windows application with control will retain control as long as messages are waiting in its queue. There is a small wrinkle to this rule: Windows considers WM_PAINT and WM_TIMER messages to be low-priority messages. If App A calls GetMessage and has only low-priority messages in its queue, Windows will pass control to an application with high-priority messages waiting.

It is important to note that an application can retain control of the CPU indefinitely if it does not call GetMessage, PeekMessage, or WaitMessage. However, an application that fails to periodically make one of these three calls is very impolite, because it is not allowing Windows to multitask.

Background Processing

In the Windows environment, an application can perform a lengthy processing task and allow Windows to multitask by using either a PeekMessage loop or a Windows timer. In both cases, the general idea is the same: break the large processing task into small pieces, processing one piece at a time and offering to yield at regular short intervals.

An application can use a PeekMessage loop to perform background processing because, unlike GetMessage, PeekMessage does not wait for a message to be placed in the application's message queue before returning. If there are no messages waiting, PeekMessage returns FALSE.

A typical PeekMessage loop resembles the following:

```
while (bDoingBackgroundWork)
   {
   if PeekMessage(&msg, hWnd, 0, 0, PM_REMOVE)
      {
      TranslateMessage(&msg); // a message is ready - process it
      DispatchMessage(&msg);
      }
   else
      {
      // PeekMessage returned FALSE, which means no messages ready.
      // Do a small piece of background work here.
      }
   }
```

The PM_REMOVE flag must be specified in the wRemoveMsg parameter; otherwise, the application will never yield. PeekMessage will not yield as long as there are messages in the application's queue. If the application does not remove and process the messages, PeekMessage may send the application into an infinite loop.

An application creates a Windows timer with the SetTimer function. An application can use a Windows timer to do background processing by setting a timer and doing a small piece of the background job each time a WM_TIMER message is received or the timer notification function is called. Even if the application is receiving a steady stream of WM_TIMER messages, other applications will still have an opportunity to run because Windows considers WM_TIMER messages to be a low priority.

It may be necessary to consider two other facts about Windows timers: they are not asynchronous, and the highest resolution that can be obtained is about 55 milliseconds.

For more information about background processing in applications running in the Windows environment, including references to sample source code, query on the following words in the Microsoft Knowledge Base:

```
   prod(winsdk) and backproc
```

For more information on using Windows timers, see Chapter 5, "The Timer," in Charles Petzold's book "Programming Windows 3" (Microsoft Press).

Additional reference words: 3.00 3.0 backproc
KBCategory:
KBSubcategory: UsrMsg

INF: GetInputState Is Faster Than GetMessage or PeekMessage
Article ID: Q35605

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

This article describes a method to quickly determine whether an application for the Microsoft Windows graphical environment has any keyboard or mouse messages in its queue without calling the GetMessage or PeekMessage functions.

The GetInputState function returns this information more quickly than GetMessage or PeekMessage. GetInputState returns TRUE (nonzero) if either a keyboard or mouse message is in the application's input queue. If the application must distinguish between a mouse and a keyboard message, GetInputState returns the value 2 for a keyboard and the value 1024 for a mouse message.

Because difficulties may arise if the application loses the input focus, use GetInputState only in tight loop conditions where execution speed is critical.

Additional reference words: 2.03 2.10 3.00 3.10 2.x yield
KBCategory:
KBSubcategory: UsrMsgRaremisc

INF: Using RegisterWindowMessage() to Communicate Between Apps

Article ID: Q66246

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The RegisterWindowMessage() function associates a unique message number with a given message name. This message number is guaranteed to be unique throughout the system but may change between Windows sessions. The new message is typically used for communication between two cooperating applications.

Using RegisterWindowMessage() is an alternative to using DDE to communicate between two or more applications. This technique is most useful if the information to be passed between applications can be contained in the wParam and/or lParam parameter of the message.

A sample application is available that demonstrates how to implement communication between two cooperating applications, using RegisterWindowMessage(). This file can be found in the Software/Data Library by searching on the word TWINS, the Q number of this article, or S12833. TWINS was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib TWINS.ZIP
KBCategory:
KBSubcategory: UsrMsgRaremisc

INF: How to Use PeekMessage Correctly in Windows

Article ID: Q74042

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Windows environment, many applications use a PeekMessage loop to perform background processing. Such applications must allow the Windows system to enter an idle state when their background processing is complete. Otherwise, system performance, "idle-time" system processes such as paging optimizations, and power management on battery-powered systems will be adversely affected.

While an application is in a PeekMessage loop, the Windows system cannot go idle. Therefore, an application should not remain in a PeekMessage loop after its background processing has completed.

More Information:

Many Windows applications use the PeekMessage function to retrieve messages while they are in the middle of a long process, such as printing, repaginating, or recalculating, that must be done "in the background." The PeekMessage function is used in these situations because, unlike the GetMessage function, it does not wait for a message to be placed in the queue before it returns.

An application should not call the PeekMessage function unless it has background processing to do between the calls to the PeekMessage function. When an application is waiting for an input event, it should call the GetMessage or WaitMessage functions.

Remaining in a PeekMessage loop when there is no background work causes system performance problems. A program in a PeekMessage loop continues to be rescheduled by the Windows scheduler, consuming CPU time and taking time away from other processes.

In enhanced mode, the virtual machine (VM) in which Windows is running will not appear to be idle as long as an application is calling the PeekMessage function. Therefore, the Windows VM will continue to receive a considerable fraction of CPU time.

Many power management methods employed on laptop and notebook computers are based on the system going idle when there is no processing to do. An application that remains in a PeekMessage loop will make the system appear busy to power management software, resulting in excessive power consumption and shortening the time that the user can run the system.

In the future, the Windows system will make more and more use of idle

time to do background processing, which is designed to optimize system performance. Applications that do not allow the system to go idle will adversely affect the performance of these techniques.

All these problems can be avoided by calling the PeekMessage function only when there is background work to do, and calling the GetMessage or WaitMessage functions when there is no background work to do.

For example, consider the following PeekMessage loop. If there is no background processing to do, this loop will continue to run without waiting for messages, preventing the system from going idle and causing the negative effects described above.

```
  // This PeekMessage loop will NOT let the system go idle.
  for (;;)
    {
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
      if (msg.message == WM_QUIT)
         return TRUE;

      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }

    BackgroundProcessing();
    }
```

This loop can be rewritten in two ways, as shown below. Both of the following PeekMessage loops have two desirable properties:

- They process all input messages before performing background processing, providing good response to user input.

- The application "idles" (waits for an input message) when no background processing needs to be done.

Improved PeekMessage Loop 1

```
  // Improved PeekMessage loop.
  for (;;)
    {
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
      if (msg.message == WM_QUIT)
         return TRUE;

      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }

    if (IfBackgroundProcessingRequired())
      BackgroundProcessing();
    else
      WaitMessage(); // Will not return until a message is posted.
```

```
    }
```

Improved PeekMessage Loop 2

```
  // Another improved PeekMessage loop
  for (;;)
    {
    for (;;)
      {
      if (IfBackgroundProcessingRequired())
        {
        if (!PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
           break;
        }
      else
        GetMessage(&msg, NULL, 0, 0, 0);

      if (msg.message == WM_QUIT)
        return TRUE;

      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }
    BackgroundProcessing();
    }
```

Note that calls to functions such as IsDialogMessage and TranslateAccelerator can be added to these loops as appropriate.

There is one case in which the loops above need additional support: if the application waits for input from a device (for example, a fax board) that does not send standard Windows messages. For the reasons outlined above, a Windows application should not use a PeekMessage loop to continuously poll the device. Rather, implement an interrupt service routine (ISR) in a dynamic-link library (DLL). When the ISR is called, the DLL can use the PostMessage function to inform the application that the device requires service. DLL functions can safely call the PostMessage function because the PostMessage function is reentrant.

For more information about background processing in applications running in the Windows environment, including references to sample source code, query this knowledge base on the following words:

   prod(winsdk) and backproc

Additional reference words: 3.00 3.10 3.x backproc
KBCategory:
KBSubcategory: UsrMsgGetmsg

INF: Sample Code Demonstrates Background Processing
Article ID: Q81139

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There are often times when it is necessary for an application to execute a time consuming process such as printing, file I/O, and math calculations in the background. Doing so allows the Windows environment to continue to service other applications while completing a lengthy process. BKGND is a file in the Software/Data Library that demonstrates using a PeekMessage loop to perform background processing.

BKGND can be found in the Software/Data Library by searching on the word BKGND, the Q number of this article, or S13302. BKGND was archived using the PKware file-compression utility.

More Information:

The BKGND sample application uses the WaitMessage function, which allows the application to "sleep" when the background process is complete. In the sample, the user determines when background processing starts and ends. In a production application, an application would probably determine when background processing was required.

For more information about background processing in applications running in the Windows environment, including references to sample source code, query this knowledge base on the following words:

   prod(winsdk) and backproc

Additional reference words: 3.00 softlib BKGND.ZIP backproc
KBCategory:
KBSubcategory: UsrMsg

INF: Win 3.0 Icon WM_GETTEXT and WM_SETTEXT Docs Incomplete
Article ID: Q69754

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The information on the WM_GETTEXT and WM_SETTEXT messages in version 3.0 of the "Microsoft Windows Software Development Kit Reference Volume 1" is incomplete.

The following information should be added to the documentation for WM_SETTEXT on pages 6-100 and 6-101:

> When the WM_SETTEXT message is sent to a static control with the SS_ICON style, LOWORD(lParam) should contain a handle to an icon, and HIWORD(lParam) should be 0 (zero). The static control will then display the new icon.

The following information should be added to the documentation for WM_GETTEXT on page 6-64:

> When the WM_GETTEXT message is sent to a static control with the SS_ICON style, the handle to the icon will be returned in the first two bytes of the buffer pointed to by the lParam. This is true only if the icon has been set using the WM_SETTEXT message.

Note: The technique described above does not work under Windows 3.1. Windows 3.1 defines the STM_GETICON and STM_SETICON messages to allow an application to change the icon associated with an icon resource.

More Information:

The following code demonstrates retrieving and changing the icon associated with an icon resource under Windows versions 3.0 and 3.1.

```
HWND    hWndStaticIcon;  // A static control having the SS_ICON style.
HICON   hIconOld;        // The icon that is currently drawn in the
                         // static control.
HICON   hIconNew;        // The new icon for the static control.
WORD    wVersionWindows;
...

wVersionWindows = LOWORD(GetVersion());

if (LOBYTE(wVersionWindows) >= 3 && HIBYTE(wVersionWindows) >= 10)
   // Windows version 3.1 or later.
   {
   // Retrieve the HICON for the currently displayed icon.
   hIconOld = (HICON) SendMessage(hWndStaticIcon, STM_GETICON, 0, 0L);
```

```
   // Specify the new icon to be drawn.
   SendMessage(hWndStaticIcon, STM_SETICON, hIconNew, 0L);
   }
else
   { // Versions of Windows earlier than 3.1.
   // Retrieve the HICON for the currently displayed icon. Note that
   // this technique will work only if this icon was previously set by
   // sending a WM_SETTEXT message to the static control.
   SendMessage(hWndStaticIcon, WM_GETTEXT, 0,
                      (LONG)(HICON FAR *)&hIconOld);

   // Specify the new icon to be drawn in the static control.
   SendMessage(hWndStaticIcon, WM_SETTEXT, 0, MAKELONG(hIconNew, 0));
   }
```

Additional reference words: 3.00 MICS3 R5.1 docerr
KBCategory:
KBSubcategory: UsrMsgRaremisc

INF: Differences Between PostAppMessage and PostMessage Funcs

Article ID: Q35774

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The following information describes the differences between the PostAppMessage and PostMessage functions, and the circumstances under which each should be used.

In most cases, the PostMessage function should be used. Essentially, both functions accomplish the same result, but PostMessage uses a window handle, and PostAppMessage uses a task handle to identify the destination window for the message.

In the Windows environment, it is possible to create a task that does not have a window associated with it, but not vice versa. Under some circumstances an application must send a message to a windowless application; there is no window handle to use as a parameter to PostMessage. In this case, use the PostAppMessage function to send the message using the task handle.

However, if an application will receive messages sent by PostAppMessage, its message loop must be modified. When a message is posted by PostAppMessage and retrieved by GetMessage, the hwnd field of the MSG structure is NULL because no window was specified as the target for the message. Therefore, it is important to process this special case and to perform whatever processing is appropriate. An application must not pass an MSG structure with a NULL hwnd field to the DispatchMessage function. Doing so will cause an error. Some possibilities for handling this special case are listed below:

- Process the message within the message loop.

- Pass the message to another procedure that is set up to process special cases.

- Set the hwnd field of the MSG structure to the window handle of the window that should receive the message.

- Change any of the other values in the MSG structure before passing the message along.

The following code demonstrates processing a message posted by the PostAppMessage function. After the window is drawn, pressing the left mouse button in the window generates a PostMessage call, and pressing the right mouse button generates an identical PostAppMessage call. When the application processes the message posted by PostAppMessage, it beeps the speaker and then passes the message to the main window.

```
//*******************************************************************
// MinWin - PostMessage versus PostAppMessage example.

#include <windows.h>

char szAppName[] = "MinWin";
HWND hMainWnd;

long FAR PASCAL WndProc(HWND, unsigned, WORD, LONG);

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
{
   MSG msg;
   WNDCLASS wndclass;

   if (!hPrevInstance)
      {
      wndclass.style = CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc = WndProc;
      wndclass.cbClsExtra = 0;
      wndclass.cbWndExtra = 0;
      wndclass.hInstance = hInstance;
      wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
      wndclass.hbrBackground = COLOR_WINDOW + 1;
      wndclass.lpszMenuName = NULL;
      wndclass.lpszClassName = szAppName;

      if (!RegisterClass(&wndclass))
         return FALSE;
      }

   hMainWnd = CreateWindow(szAppName, szAppName, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
        NULL, NULL, hInstance, NULL);

   ShowWindow(hMainWnd, nCmdShow);
   UpdateWindow(hMainWnd);

   while (GetMessage(&msg, NULL, 0, 0))
      {
      if (msg.hwnd == NULL) // Beep, then pass the message to
         {                  //  the window
         MessageBeep(0);
         msg.hwnd = hMainWnd;
         }

      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }

   return msg.wParam;
}

long FAR PASCAL WndProc(HWND hWnd, unsigned iMessage,
                        WORD wParam, LONG lParam)
```

```
{
   HANDLE hInst;
   HANDLE hTask;

   hInst = GetWindowWord(hWnd, GWW_HINSTANCE);

   switch (iMessage)
      {
   case WM_LBUTTONDOWN:
      PostMessage(hWnd, WM_USER+0x1000, 0, 0L);
      break;

   case WM_RBUTTONDOWN:
      PostAppMessage(GetWindowTask(hWnd), WM_USER+0x1000, 0, 0L);
      break;

   case WM_USER+0x1000:
      MessageBox(NULL, "Message Received", "WM_USER", MB_OK);
      break;

   case WM_DESTROY:
      if (hWnd == hMainWnd)
         PostQuitMessage(0);
      break;

   default:
      return DefWindowProc(hWnd, iMessage, wParam, lParam);
      }

   return 0L;
}
```

Additional reference words: 2.03 2.10 3.00 3.10 2.x
KBCategory:
KBSubcategory: UsrMsgPostmsg

INF: Background Processing with PeekMessage Code Example

Article ID: Q71670

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

BACKGRND is a file in the Software/Data Library that demonstrates using a PeekMessage loop to perform background processing in an application for the Windows environment. BACKGRND can be found in the Software/Data Library by searching on the word BACKGRND, the Q number of this article, or S13017. BACKGRND was archived using the PKware file-compression utility.

More Information:

BACKGRND uses a PeekMessage loop to do background processing. The background task is to draw rectangles at random in the application's client window. BACKGRND draws one "batch" of rectangles each time through the PeekMessage loop, and all the rectangles in each batch are drawn in the same color. This provides a visual indication of how much background work is done before BACKGRND offers to yield.

Both the "batch size" and the total number of rectangles drawn can be changed using a menu selection.

BACKGRND does not replace the main GetMessage loop with a PeekMessage loop. Instead, it enters a PeekMessage loop only when the user chooses the Draw! menu item.

While it draws the rectangles, BACKGRND displays a modeless dialog box that:

1. Allows the user to cancel the background processing before all the rectangles have been drawn (by choosing the Cancel button)

2. Indicates both the total number of rectangles to be drawn and the batch size

3. Displays a "gas gauge" that indicates the percentage of the background task that has been completed

BACKGRND can be run along with another Windows application (for example, Notepad) to illustrate the difference between "polite" and "impolite" background processing.

If the batch size is set to a "polite" small value (for example, 10 rectangles each time through the PeekMessage loop), the user can easily type within Notepad while BACKGRND is drawing rectangles. BACKGRND draws rectangles in the spare time slices between the messages generated by the user's keystrokes, and calls PeekMessage

often enough that the user will still find Notepad responsive.

If the batch size is set to an "impolite" large value (for example, 1000 rectangles each time through the PeekMessage loop), BACKGRND can complete its background job much more quickly, because there are fewer calls to PeekMessage and thus less overhead. However, a user trying to type in Notepad will find the keyboard so unresponsive that Notepad will be essentially useless.

For more information about background processing in applications running in the Windows environment, including references to sample source code, query on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and backproc

Additional reference words: 3.00 3.0 softlib BACKGRND.ZIP backproc
KBCategory:
KBSubcategory: UsrMsgBackgrnd

INF: Performing Background Processing Without Using Timers

Article ID: Q36324

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A Windows application that performs a long, background task, such as repaginating a word processing document, can be designed in a number of different ways.

A polling task can be accomplished by setting a timer to fire at the desired interval. Many nonpolling tasks can be performed in pieces. Although Windows does not have a method to schedule processing based on overall system load, an application can wait until there are no other messages to be processed by that application before proceeding. This article discusses the code required to implement this method.

More Information:

It is important that each piece of the task be relatively small. This allows Windows to devote processing time to other applications running in the system. Similarly, once the task is complete, it is important that the application signal that it is idle. This allows Windows to optimize its performance and to prolong battery life on portable computers.

The following code skeleton demonstrates how this might be implemented:

```
WinMain
{
do application initialization

fBackgroundToDo = TRUE;
fRunning = TRUE;

while (fBackgroundToDo && fRunning)
     {
     if (fBackgroundToDo)
          {
          if ((PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
               {
               if (msg.message == WM_QUIT)
                    {
                    fRunning = FALSE;
                    break;
                    }

               TranslateMessage(&msg);
               DispatchMessage(&msg);
```

```
          }
     else
          fBackgroundToDo = DoABitOfBackground();
     }
else if ((fRunning = GetMessage(&msg, NULL, NULL, NULL)) != 0)
     {
     TranslateMessage(&msg);
     DispatchMessage(&msg);

     fBackgroundToDo = IsThereBackgroundToDo();
     }
}
}
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMsgBackgrnd

INF: When PostMessage() Will Return 0, Indicating Failure

Article ID: Q36584

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

PostMessage(hWnd,wMsg,wParam,lParam) returns 0, indicating failure, for one of the following two reasons:

1. The window handle (hWnd parameter to PostMessage()) receiving the message is -1. If this is the case, the message (wMsg) is broadcast to all windows.

2. The window to receive the message has a full application queue.

Note that PostMessage() does not return NULL if the hWnd parameter in the PostMessage() call is an invalid window handle. Also note that IsWindow() can be called to verify whether a given handle is indeed a valid window handle.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMsgPostmsg

INF: Defining Private Messages for Application Use

Article ID: Q86835

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows environment, an application can define a private message for its own use without calling the RegisterWindowMessage function. Message numbers between 0x8000 and 0xBFFF are reserved for this purpose.

More Information:

On page 206 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 3: Messages, Structures, and Macros" manual for version 3.1, the documentation for the WM_USER message lists four ranges of message numbers as follows:

Message Number	Description
0 through WM_USER-1	Messages reserved for use by Windows.
WM_USER through 0x7FFF	Integer messages for use by private window classes.
0x8000 through 0xBFFF	Messages reserved for use by Windows.
0xC000 through 0xFFFF	String messages for use by applications.

When an application subclasses a predefined Windows control or provides a special message in its dialog box procedure, it cannot use a WM_USER+x message to define a new message because the predefined controls use some WM_USER+x messages internally. It was necessary to use the RegisterWindowMessage function to retrieve a unique message number between 0xC000 and 0xFFFF.

To avoid this inconvenience, messages between 0x8000 and 0xBFFF were redefined to make them available to an application. Messages in this range do not conflict with any other messages in the system. In the next release of the Windows Software Development Kit (SDK), the WINDOWS.H header file will define the constant WM_APP to be the beginning of the range of private messages (0x8000).

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrMsgRaremisc

INF: Some Basic Concepts of a Message-Passing Architecture
Article ID: Q74476

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The following is excerpted from an article in the April 1991 issue of Software Design (Japan).

More Information:

Asynchronous message-passing means that Windows will send an application messages to act on and that these messages may come in any order. At present, all messages are sent to specific windows. Every window has a function that Windows calls to send that window a message. This function processes the message and returns to Windows. When the function returns to Windows, Windows may then send messages to other windows in the same program, other programs, or to the same window again.

Because most of these messages are generated by user actions (picking an item in a menu, moving a window, and so forth), the specific messages a window receives will differ each time the program is run. This is what makes the messages asynchronous.

This message passing is what makes Windows programming difficult for many programmers. The programmer is no longer writing a program in which he or she controls the flow from beginning to end. Rather, a Windows program is written as a large number of objects, each one designed to handle a specific message from Windows.

Understanding message passing is critical and because it leads to so much confusion, the concept will be explained in greater detail in this article. If message passing is understood, the remainder of Windows can be learned fairly easily. However, it is very unlikely that a Windows program or other graphical user interface (GUI) program can be successfully developed without a thorough understanding of message passing.

In a message-passing system, the focus changes from being proactive (the programmer controls the program flow) to being reactive (Windows controls the program flow). [Or as it has been put by some Macintosh programmers, "Don't call us, we'll call you."] For example, consider the situation where a user chooses an action from a menu in a program. In a proactive program, the program reads the keyboard, determines that the key(s) pressed are meant to run the action, and calls the function that performs that action. In a reactive system, the program is sent a message indicating that the user chose that item from a menu. When the program receives the message, it calls the function that performs the action. When this function is done, control returns

to the system. Although the reactive approach is substantially different from the proactive approach, it is also simpler.

In Windows, every window (including dialog boxes) has a "response function" registered to it. When Windows sends a message to a window, it calls the response function for that window and passes it the message. All messages from Windows are passed to window response functions; there is no other way for Windows to send a message to a program. Therefore, all messages are for a specific window or group of windows.

However, there are four considerations involved with this method:

1. Messages are sent in two distinct ways. The first method consists of messages that are posted to a first-in, first-out queue (PostMessage). The second method consists of messages that are sent (SendMessage). Posted messages, aside from PAINT messages, are serialized, meaning that messages cannot be posted anywhere except to the end of the queue, and the application is sent messages only from the beginning of the queue. Posted PAINT messages are an exception. They are added together and sent only when there is nothing else in the queue. This is done to reduce the number of times a window has to paint itself.

   Messages that are sent are passed to the application immediately, and the send function does not return until the message is processed. However, when a message is posted, an indeterminate number of messages and amount of time will pass before the message is actually sent to a window and acted on. Also, when a message is sent, an message posted earlier may not yet have been acted on.

2. Sending messages or calling functions (which may, as part of their actions, also send messages), can lead to additional messages being generated. The most dangerous situation is where the action for a message generates the same message again. If, while processing a message, an application sends the same message to itself, the application will run out of stack space quite quickly. If an application POSTS the same message to itself, the application will not run out of stack but it will generate an unending stream of messages.

3. If, while processing a message, an application calls a function that sends a message, the application will process the second message in the middle of processing the first message. Therefore, each window response function MUST be fully re-entrant. It is even possible for a function to be re-entered to process the same message as the message currently being processed. For this reason, using global or static variables in a response function is very dangerous.

   Also, if the application uses properties, scratch files and/or other data storage mechanisms, extreme caution is required. Consider the situation where one message reads in data from a file, then a second message reads in the same data, makes changes to that data, and writes it back to the file. If the first message makes additional changes to the data and writes its new data back to the file, the changes caused by the second message are completely lost.

With files, each application must implement its own sharing mechanism. For properties, allocated memory and other memory storage, there is a simple solution: lock the item and use the pointer returned. Because only one lock is allowed, this prevents contention. Never copy the data into a scratch buffer to copy back later.

4. Because messages come in due to user interaction, the application cannot be written to assume that when a particular message is received that another message has already been processed and performed its functions. While, for a given action, a specific sequence of messages may occur, in the interest of remaining completely compatible with potential changes in future versions of Windows, it is recommended that no message ordering dependencies be introduced unless absolutely necessary. The best example of this is that the first message a window gets when it is being created is not WM_CREATE, rather it is WM_GETMINMAXINFO. When one message must logically follow a second (such as WM_CREATE always preceding WM_DESTROY), then it is fine to depend on the specific ordering of those two messages.

The asynchronous, reactive nature of Windows programming can cause confusion. Because the program has no control over the order that messages arrive, the response to ANY specific message CANNOT depend on other messages having been processed or NOT been processed.

To confuse matters even further, an application may be in the middle of processing one message when it calls a Windows function that sends the application another message. When processing this second message, some dependent processing may be only half finished. If an application will check and only do some processing if another message has not already performed it, the application must be prepared for the case where another message has begun the processing, but has not completed it.

Do not be confused into thinking that a Windows task is preemptively multitasked; it is not. As a matter of fact, Windows is non-preemptive between all of its Windows tasks. Therefore, an application generally does not need to be designed to process a user-originated message in the middle of processing another message. However, when an application calls a Windows function, the application may then get a set of specific messages sent to it by Windows before the called function returns.

Further, when an application sets up a modal dialog box, the DialogBox() function will not return until after the dialog box is dismissed and processing completed. Therefore, after calling the dialog box function, all combinations of user-generated messages may be received before the function returns.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMsgRaremisc

INF:
Article ID: Q74528

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Power-managed personal computers, such as some battery-powered notebook computers, are designed to conserve power during periods of CPU inactivity or "idle time." Although most power-saving measures should occur transparently to an application, developers should be aware of several attributes that make an application "power friendly."

To qualify as "power-friendly," an application must use the PeekMessage and Yield functions carefully, preferably only for temporary background tasks. As long as an application is in a PeekMessage loop, the Windows system cannot go idle. Therefore, an application should not remain in a PeekMessage loop after its background processing has completed.

More Information:

Traditionally, an application running in the Windows multitasking environment needed only to ensure that once it was finished with its processing it yielded the processor to other applications in the system. However, the increased popularity of battery-powered systems imposes another responsibility on applications: to allow Windows to "go idle" as often as possible.

To Windows, the system appears idle when all executing programs are awaiting input. All applications have called the GetMessage or WaitMessage function, and no pending messages are in any application queue. At such times, the Windows kernel broadcasts an idle notification by using an MS-DOS interrupt. A power-managed PC can use this notification as a signal to take power-saving measures.

When an application is in a PeekMessage or Yield loop, it prevents Windows from going idle, thus thwarting attempts by power-managed PCs to save power.

An application's main message handler should use a GetMessage loop or equivalent structure (such as the combined use of the PeekMessage and WaitMessage functions) when no background processing is required. This will allow Windows to go idle and power-saving measures to take effect.

Applications that must periodically service a task (for example, a communications connection) should poll, in response to a message from a timer, if there is no other way to determine when the status of the device changes.

For more information about PeekMessage loops and idle time, query the Microsoft Knowledge Base on the following words:

```
prod(winsdk) and peekmessage and idle
```

For more information about background processing in applications running in the Windows environment, including references to sample source code, query the Microsoft Knowledge Base on the following words:

```
prod(winsdk) and backproc
```

Additional reference words: 3.00 3.10 APM backproc
KBCategory:
KBSubcategory: UsrMsgGetmsg

INF: Posting Frequent Messages Within an Application

Article ID: Q40669

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The object-oriented nature of Windows programming can create a situation in which an application posts a message to itself. When such an application is designed, care must be taken to avoid posting messages so frequently that system messages to the application are not processed. This article discusses two methods of using the PeekMessage function to combat this situation.

More Information:

In the first method, a PeekMessage loop is used to check for system messages to the application. If none are pending, the SendMessage function is used from within the PeekMessage loop to send a message to the appropriate window. The following code demonstrates this technique:

```
while (fProcessing)
     {
     if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
          if (msg.message == WM_QUIT)
               break;
          /* process system messages */
          }
     else
          {
          /* perform other processing */
          ...
          /* send WM_USER message to window procedure */
          SendMessage(hWnd, WM_USER, wParam, lParam);
          }
     }
```

In the second method, two PeekMessage loops are used, one to look for system messages and one to look for application messages. PostMessage can be used from anywhere in the application to send the messages to the appropriate window. The following code demonstrates this technique:

```
while (fProcessing)
     {
     if (PeekMessage(&msg, NULL, 0, WM_USER-1, PM_REMOVE))
          {
          if (msg.message == WM_QUIT)
               break;
```

```
          /* process system messages */
          }
     else if (PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_REMOVE))
          /* process application messages */
     }
```

An application should use a PeekMessage loop for as little time as possible. To be compatible with battery-powered computers and to optimize system performance, every Windows application should inform Windows that it is idle as soon and as often as possible. An application is idle when the GetMessage or WaitMessage function is called and no messages are waiting in the application's message queue.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrMsgPostmsg

PRB: PeekMessage(hWnd==-1) Causes Fatal Exit 0x0007

Article ID: Q74701

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

When the PeekMessage function is called with an hWnd parameter of -1, no messages are returned by the retail version of Windows, and the debugging version of Windows exits with fatal exit 0x0007 (Invalid Window Handle). According to page 4-330 of the "Microsoft Windows Software Development Kit Reference Volume 1," this call should return only messages with an hWnd of NULL that are posted by the PostAppMessage function.

CAUSE

Microsoft has confirmed that the documentation is incorrect. The PeekMessage function should not be called with a hWnd parameter of -1.

RESOLUTION

Calling PeekMessage with a hWnd parameter of NULL will retrieve messages that were posted with the PostAppMessage function.

Microsoft has confirmed that this documentation error has been corrected on page 739 of the "Microsoft Windows Software Development Kit Programmer's Reference Volume 2: Functions" for version 3.10.

Additional reference words: docerr 3.00
KBCategory:
KBSubcategory: UsrMsgGetmsg

INF: Using PeekMessage() Loops in a Dialog Box

Article ID: Q74795

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When a modal dialog box is created, Windows uses its own message loop, which is stored in USER. If the application takes responsibility for processing messages [using a PeekMessage() loop], then IsDialogMessage() must be used to pass appropriate messages to the dialog box.

The following code fragment demonstrates the correct technique:

```
  bFlag = TRUE;

  while ((bFlag) || (PeekMessage(&message, hDlg, 0, 0, PM_REMOVE)))
    {
    if (!IsDialogMessage(hDlg, &message))
      {
      TranslateMessage(&message);
      DispatchMessage(&message);
      }
    }
```

If technique is not used, accelerator keys in the dialog box will not function properly.

For some caveats as to the proper use of PeekMessage() loops, please query on the following words:

   prod(winsdk) and peekmessage and idle

Additional reference words: 3.0 3.00
KBCategory:
KBSubcategory: UsrMsgGetmsg

INF: Handling WM_QUIT While Not in Primary GetMessage() Loop

Article ID: Q89738

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application can terminate while in a message loop that is different from the primary GetMessage() loop. If this secondary message loop retrieves the WM_QUIT message from the message queue, the message must be reposted so that the primary GetMessage() loop can exit.

More Information:

An application can get into a secondary message loop that is different from the primary GetMessage() loop in order to track drag messages, yield periodically during lengthy processing, and so on. The user may be able to terminate the application while the secondary message loop is being used to retrieve messages from the message queue.

If the GetMessage()/PeekMessage() call used in the secondary loop specifies a message filter that does not include WM_QUIT, WM_QUIT will not be retrieved from the message queue. When the loop terminates, control will be returned to the primary GetMessage() loop, which will retrieve the WM_QUIT message.

If the GetMessage()/PeekMessage() call in the secondary loop retrieves all messages or specifies a message filter that includes the WM_QUIT message, the WM_QUIT message that is retrieved must be reposted and control must be returned to the primary message loop. The primary message loop will then retrieve the reposted WM_QUIT message and exit.

```
while (bNotDone)
{
   // Secondary message loop
   while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
   {
      if (msg.message == WM_QUIT)
      {
         // Repost the QUIT message so that it will be retrieved by the
         // primary GetMessage() loop.
         PostQuitMessage(msg.wParam);
         return FALSE;
      }
      // Pre-process message if required (TranslateAccelerator etc.)
      TranslateMessage(&msg);
      DispatchMessage(&msg);
      :
      :
   }
   :
```

}

The primary message loop will not exit if the secondary message loop does not repost the WM_QUIT message. Consequently, the application will remain in memory even though all its windows are destroyed.

An alternative solution is to prevent the user from terminating the application while the application is in the secondary message loop.

Additional reference words: 3.00 3.10 3.0 3.1 close
KBCategory:
KBSubcategory: UsrMsgRaremisc

INF: Nonzero Return from SendMsg() with HWND_BROADCAST
Article ID: Q102588

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

SUMMARY
=======

The SendMessage() function calls the window procedure for the specified window and does not return until that window has processed the message and returned a value. Applications can send messages to all top-level windows in the system by specifying HWND_BROADCAST as the first parameter to the SendMessage() function. In doing so, however, applications lose access to the return values resulting from the SendMessage() call to each of the top-level windows.

MORE INFORMATION
================

When a call to SendMessage() is made, the value returned by the window procedure that processed the message is the same value returned from the SendMessage() call.

Among other things, SendMessage() determines whether the first parameter is HWND_BROADCAST (defined as -1 in WINDOWS.H). If HWND_BROADCAST is the first parameter, SendMessage enumerates all top-level windows in the system and sends the message to all these windows. Because this one call to SendMessage() internally translates to a number of SendMessage() calls to the top-level windows, and because SendMessage() can return only one value, Windows ignores the individual return values from each of the top-level window procedures, and just returns a nonzero value to the application that broadcast the message. Thus, applications that want to broadcast a message to all top-level windows, and at the same time expect a return value from each SendMessage() call, should not specify HWND_BROADCAST as the first parameter.

There are a couple of ways to access the correct return value from messages sent to more than one window at a time:

- If the broadcasted message is a user-defined message, and only a few other applications respond to this message, then those applications that trap the broadcasted message must return the result by sending back another message to the application that broadcast the message. The return value can be encoded into the message's lParam.

- If the application does not have control over which application(s) will respond to the message, and it still expects a return value, then the application must enumerate all the windows in the system using EnumWindows() function, and send the message separately to each window it obtained in the enumeration callback function.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrMsgPostmsg

INF: Message Retrieval in a DLL

Article ID: Q96479

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

When a function in a dynamic-link library (DLL) retrieves messages on behalf of a calling application, the following must be addressed:

- The application and the DLL may be re-entered.
- The application can terminate while in the DLL's message retrieval loop.
- The DLL must allow the application to preprocess any messages that may be retrieved.

More Information:

The following concerns arise when a function in a DLL retrieves messages by calling GetMessage or PeekMessage:

- When the DLL function retrieves, translates, and dispatches messages, the calling application and the DLL function may be re-entered. This is because message retrieval can cause the calling application to respond to user input while waiting for the DLL function to return. The DLL function can return a reentrancy error code if this happens. To prevent reentrancy, disable windows and menu-items, or use a filter in the GetMessage or PeekMessage call to retrieve specific messages.

- The application can terminate while execution is in the DLL function's message retrieval loop. The WM_QUIT message retrieved by the DLL must be re-posted and the DLL function must return to the calling application. This allows the calling application's message retrieval loop to retrieve WM_QUIT and terminate.

- When the DLL retrieves messages, it must allow the calling application to preprocess the messages (to call TranslateAccelerator, IsDialogMessage, and so forth) if required. This is be done by using CallMsgFilter to call any WH_MSGFILTER hook that the application may have installed.

The following code shows a message retrieval loop in a DLL function that waits for a PM_COMPLETE message to signal the end of processing:

```
while (notDone)
{
   GetMessage(&msg, NULL, 0, 0);

   // PM_COMPLETE is a WM_USER message that is posted when
   // the DLL function has completed.
```

```
   if (msg.message == PM_COMPLETE)
   {
      Clean up and set result variables;
      return COMPLETED_CODE;
   }
   else if (msg.message == WM_QUIT)  // If application has terminated...
   {
      // Repost WM_QUIT message and return so that calling
      // application's message retrieval loop can exit.
      PostQuitMessage(msg.wParam);
      return APP_QUIT_CODE;
   }

   // The calling application can install a WH_MSGFILTER hook and
   // preprocess messages when the nCode parameter of the hook
   // callback function is MSGF_MYHOOK. This allows the calling
   // application to call TranslateAccelerator, IsDialogMessage, etc.
   if (!CallMsgFilter(&msg, MSGF_MYHOOK))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   :
   :
}
```

Define MSGF_HOOK to a value greater than or equal to MSGF_USER defined in WINDOWS.H to prevent collision with values used by Windows.

Preprocessing Messages in the Calling Application

The calling application can install a WH_MSGFILTER hook to preprocess messages retrieved by the DLL. It is not required for the calling application to install such a hook if it does not want to preprocess messages.

```
   lpfnMsgFilterProc = MakeProcInstance((FARPROC)MsgFilterHookFunc,
                                         ghInst);
   hookprocOld = SetWindowsHook(WH_MSGFILTER, lpfnMsgFilterProc);
   // Call the function in the DLL.
   DLLfunction();
   UnhookWindowsHook(WH_MSGFILTER, lpfnMsgFilterProc);
   FreeProcInstance(lpfnMsgFilterProc);
```

MsgFilterHookFunc is the hook callback function:

```
LRESULT CALLBACK MsgFilterHookFunc(int nCode, WPARAM wParam,
                                   LPARAM lParam)
{
   if (nCode < 0)
      return DefHookProc(nCode, wParam, lParam, &hookprocOld);

   // If CallMsgFilter is being called by the DLL.
   if (nCode == MSGF_MYHOOK)
   {
```

```
      Preprocess message (call TranslateAccelerator,
          IsDialogMessage etc.);
      return 0L if the DLL is to call TranslateMessage and
          DispatchMessage. Return 1L if TranslateMessage and
          DispatchMessage are not to be called.
   }
   else return 0L;
}
```

Additional reference words: 3.10 yield
KBCategory:
KBSubcategory: UsrMsgGetmsg

INF: Split Scrolling Using Two Windows

Article ID: Q19736

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To create a window that has the following characteristics:

 - a parent window that is a tiled window with both horizontal and vertical scroll bars

 - a child window that is a title, one line long, across the top of the parent window, and is stationary (that is, it does not scroll off the screen when the tiled window is scrolled vertically)

create two child windows:

 - The first should be one line.

 - The second should be the rest of the client area.

Then perform all of the scrolling in the second child.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrPntScrolling

INF: CS_SAVEBITS Class Style Bit

Article ID: Q31073

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

If the CS_SAVEBITS style is included when registering a popup window, a bitmap copy of the screen image that the window will obscure is saved in memory when the window is displayed.

The bitmap will be redisplayed at its original location and no WM_PAINT messages will be sent to the obscured windows if the following is true when the window is removed from the display:

1. The memory used by the saved bitmap has not been discarded.

2. Other screen actions have not invalidated the image that has been stored.

As a general rule, this bit should not be set if the window will cover more than half the screen; a lot of memory is required to store color bitmaps.

The window will take longer to be displayed because memory needs to be allocated. The bitmap also needs to be copied over each time the window is shown.

Use should be restricted to small windows that come up and are then removed before much other screen activity takes place. Any memory calls that will discard all discardable memory, and any actions that take place "under" the window, will invalidate the bitmap.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrPntRareMisc

INF: Using EndPaint() and BeginPaint()

Article ID: Q10216

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The PAINTSTRUCT structure contains information for an application that can be used to paint the client area of a window owned by that application.

The PAINTSTRUCT data structure is automatically updated by the BeginPaint() Windows procedure; if a window has been restored, BeginPaint() will update the PAINTSTRUCT fields (for example, fRestore). The application would be duplicating effort if it also updated the fields of the PAINTSTRUCT after a window is restored. The EndPaint() procedure is required after BeginPaint(). The application only needs to be concerned about its own paint procedure that comes between BeginPaint() and EndPaint().

See page 7-55 in the "Microsoft Windows Software Development Kit Reference Volume 2" for more information on the PAINTSTRUCT data structure.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrPntBeginpnt

INF: Changing Window Colors with Control Panel

Article ID: Q11246

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

In Windows, changing the screen background color or the test color
will affect the following programs:

```
   Notepad        MS-DOS Executive
   Clock          PIF Editor
   Control        Spooler Panel
   Calendar       Terminal
   Clipboard      Write
```

However, the following programs are not affected:

```
   Calculator     Reversi
   Cardfile       COMMAND.COM
   Paint
```


More Information:

It is up to the application to look in the WIN.INI file to see what
[color] preferences are selected. Some applications do not change
because either they do not look in the WIN.INI file for those colors,
or the applications choose to ignore them.

For an application to be aware of system color changes, the
WM_SYSCOLORCHANGE message must be used in the Windows procedures.

For example, suppose a static hBrush is in the WndProc for repaint
purposes. To be aware of system color changes, update the brush based
on system color changes by doing something similar to the following:

```
 case WM_SYSCOLORCHANGE:
    DeleteObject(hBrush);
    hBrush = CreateSolidBrush(GetSysColor(COLOR_WINDOW));
    return 0;
```

Additional reference words: 3.1 3.10 3.0 3.00
KBCategory:
KBSubcategory: UsrPntRareMisc

INF: Panning and Scrolling in Windows
Article ID: Q11619

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When using bitmaps, the mapping mode is ignored and physical units (in other words, MM_TEXT pixels) are used. It is not necessary to use the extent/origin routines to keep track of the logical origin.

If scrolling is desired and if there are no child windows in the client area, it is best to BitBlt the client area to scroll it and PatBlt the uncovered area with the default brush.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrPntScrolling

INF: Using Blinking Text in an Application

Article ID: Q11787

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

It is possible to create blinking text in a Windows application. Because there are no character attributes similar to the normal MS-DOS text environment, the application must repeatedly paint the screen to implement the flash. This article, and an accompanying file in the Software/Data Library, demonstrate how this is done.

More Information:

A timer is used to determine the rate at which the text flashes. Timer messages are processed by inverting the appropriate area in the window using the DSTINVERT action of the PatBlt function. The second time that the PatBlt function is called, the text returns to its original state. Alternatively, the PATINVERT action of the PatBlt function may be used. To use this method, an appropriate brush must be selected into the display context as the current pattern. This method requires more work, however, it is more flexible.

The rate at which the text blinks can be set to match the cursor blink time set in the Control Panel. To do this, the following code should be run when the application starts and in response to WM_WININICHANGE messages:

```
   nRate = GetProfileInt(
     (LPSTR)"windows",          /* heading in [] */
     (LPSTR)"CursorBlinkRate",  /* string to match */
      550);                     /* default value */
```

Be sure to delete the timer when the application terminates.

There is a sample program in the Software/Data Library named BLINK that uses this technique to demonstrate blinking text. BLINK can be found in the Software/Data Library by searching on the word BLINK, the Q number of this article, or S12064. BLINK was archived using the PKware file-compression utility.

Additional reference words: TAR55086 softlib 2.03 2.10 2.x 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrPntRareMisc

INF: BeginPaint() Invalid Rectangle in Client Coordinates
Article ID: Q19963

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The BeginPaint() function returns a far pointer to a PAINTSTRUCT data structure through its second parameter. The rcPaint field of this structure specifies the update rectangle in client-area coordinates (relative to the upper-left corner of the window client area).

This update rectangle also serves as the clipping area for painting in the window, unless the invalid area of the window is expanded using the InvalidateRect() function.

Additional reference words: 3.0 2.0 3.1
KBCategory:
KBSubcategory: UsrPntBeginPnt

INF: Using the WM_CTLCOLOR Message

Article ID: Q32685

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A WM_CTLCOLOR message is sent to a window each time one of its child window controls (radio button, check box, scroll bar, and so forth) is to be painted on the screen. This message precedes the painting of the control. When it is desirable to change the appearance of controls, this can be done by processing the WM_CTLCOLOR message.

More Information:

When WM_CTLCOLOR is sent, wParam contains a handle to the display context for the child window (in this case the control). The LOWORD of lParam identifies the child window by its ID number, and the HIWORD of lParam contains one of the following values, specifying the type of control that is to be drawn:

```
   CTLCOLOR_BTN             button control
   CTLCOLOR_DLG             dialog box
   CTLCOLOR_EDIT            edit control
   CTLCOLOR_LISTBOX         list box
   CTLCOLOR_MSGBOX          message box
   CTLCOLOR_SCROLLBAR       scroll bar
   CTLCOLOR_STATIC          static text, frame, or rectangle
```

When WM_CTLCOLOR is processed, a handle to a brush must be returned. Failure to return a brush handle will result in a Windows FatalExit on the debugging monitor in the debug version of Windows.

DefWindowProc already returns a handle in response to this message; however, an application can return a different handle to customize the color of controls. The handle that is returned specifies the brush to be used to paint the control. For example, in Windows 2.x, the following code paints the background of all buttons light gray:

```
        case WM_CTLCOLOR:
            if (HIWORD(lParam) == CTLCOLOR_BTN)
                return (GetStockObject(LTGRAY_BRUSH));

            return (GetStockObject(WHITE_BRUSH));
```

In this case, the backgrounds for all other controls are painted white. Note that GetStockObject returns a handle to the stock brush specified by the parameter. To change the background color of a button control in Windows 3.0, it is necessary to create an owner draw button.

Returning a brush handle presents some interesting possibilities

because brush handles are not limited to those returned from GetStockObject. Pattern brushes can be built from bitmaps. If a pattern brush handle is returned in response to a WM_CTLCOLOR message, the brush would be used to paint the background of controls.

The following code changes the painting of a scroll bar's thumb track area to the basket-weave pattern found in Paint:

```
    /* Add these global variables. The array of WORDs specifies the */
    /* pattern for the brush                                        */
    HBRUSH  hBrush;
    HBITMAP hBitmap;
    WORD    wWeave[]={0x0F, 0x8B, 0xDD, 0xB8, 0x70, 0xE8, 0xDD, 0x8E};

     ...

    /* Add these lines to WinMain                                   */
    hBitmap = CreateBitmap(8, 8, 1, 1, (LPSTR)wWeave);
    hBrush = CreatePatternBrush(hBitmap);

     ...

   /* Add this case to the Windows procedure or wherever the       */
   /* messages are processed.                                      */
    case WM_CTLCOLOR:
        if (HIWORD(lParam) == CTLCOLOR_SCROLLBAR)
            return (hBrush);

        return (GetStockObject(WHITE_BRUSH));
```

The WM_CTLCOLOR message also applies to the following five classes of controls:

1. Check boxes, radio buttons, and push buttons: Paints the rectangular area on which control is placed with selected brush; control shape and text is drawn over the painted pattern.

2. Edit control: Paints the editing area.

3. Group box: Paints the rectangular area behind the title text.

4. Scroll bars: Paints the area around the thumb track.

5. List box: Paints the listing area.

Static text, frames, and rectangles are unaffected by WM_CTLCOLOR.

Note: In Windows 3.0, an application cannot change the color of a button face. However, the user can use the Control Panel to change the button colors for all applications in the system. This can also be accomplished by modifying the [colors] section of the WIN.INI file to add a "ButtonFace=" line that specifies the RGB color value for the button face color.

Processing the WM_CTLCOLOR message only changes the color of child windows created by an application. Windows sends the WM_CTLCOLOR

message to the parent window of each of these controls. Scroll bars that are included by Windows as a part of edit controls or list boxes are not affected. The thumb track area of system-generated scroll bars can only be changed in WIN.INI or through the Control Panel.

It is also possible to paint the entire background of a dialog box. The following code can be used to provide a dialog box color:

```
long FAR PASCAL MainWindowProc(....);

            ...

      case WM_CREATE:
         hTempBrush = LoadBitmap(hInst, (LPSTR)"MyPatternBrush");
         hBrush = CreatePatternBrush(hTempBrush);
         DeleteObject(hTempBrush);

            ...

      case WM_DESTROY:
         DeleteObject(hBrush);

            ...

BOOL FAR PASCAL MyDialogProc(....);

            ...

      case WM_CTLCOLOR:
         if (bMonoChrome)
            return (FALSE);   // Do nothing if on a monochrome monitor
                              // bMonoChrome is a global set during
                              // initialization.

         if (CTLCOLOR_DLG == HIWORD(lParam))
            UnrealizeObject(hBrush);

         SelectObject((HDC)wParam, hBrush);

         if (CTLCOLOR_DLG == HIWORD(lParam))
            SetBrushOrg((HDC)wParam, 0, 0);

         SetBkMode((HDC)wParam, TRANSPARENT);

         SetTextColor((HDC)wParam, RGB(0, 0, 0));

         return(hBrush);

            ...
```

This code will ensure that a patterned bitmap will line up correctly across the entire dialog box. "MyPatternBrush" is the name of a bitmap created using SDKPaint. This bitmap must be included in the RC file.

Additional reference words: MICS3 R5.1 2.10 3.00
KBCategory:
KBSubcategory: UsrPnt

INF: Designing Applications for High Screen Resolutions

Article ID: Q74527

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Every day, the Microsoft Windows graphical environment is run at greater screen resolutions and sizes with more colors. This article outlines several guidelines to follow to ensure that an application runs well on all video configurations.

More Information:

Avoid the following three pitfalls:

1. Hard-coded screen sizes. The VGA 640 by 480 standard resolution screen will not be around forever. If more screen area lets the application display more data, let the window grow as big as the user demands. If screen dimensions are important to the application, use the GetSystemMetrics function with the SM_CXSCREEN and SM_CYSCREEN constants to obtain the screen size.

2. Scaling all screen output proportional to resolution. Many users prefer high screen resolutions for the extra screen "real estate" they allow, not just for improved readability.

3. Centering dialog boxes relative to the screen. On a very large, high-resolution monitor, many applications can run simultaneously without overlapping. Because users expect dialog boxes and notifications to appear near the main application window, position dialog boxes relative to the main application window.

Perform the following five steps:

1. Test the application at all screen sizes. Watch out for poorly scaled objects that may require fine-tuning, oversized windows, and hard-to-read fixed-size text.

2. Take advantage of leftover screen real estate. For example, if an application document does not require the full screen, allow users the option of filling the remaining space with multiple windows, toolbars and other controls, help information, and so forth.

3. Use scaled graphical screen resources where possible. If the application uses a toolbar, icons, pictures, and so forth, Windows metafiles are likely to be more versatile than bitmaps (although perhaps slower). Scaling based on screen resolution is more effective than scaling based on application window size.

4. Use the WM_GETMINMAXINFO message to help Windows decide how big the

maximized application window needs to be. Maximizing a text editor application to the full 2048 x 2048 screen size is probably not useful. Maximizing an application at startup is particularly impolite.

5. Optimize window redraws. If a window contains a lot of information, try to redraw the entire window as infrequently as possible. Use the InvalidateRect function to coalesce window update regions into a single repaint. Even if there is not much to draw, many application windows may be open on a high-resolution screen.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrPntRareMisc

FIX: TextOut Draws Dotted Underscore with Most Colored Text

Article ID: Q74740

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

SYMPTOMS
========

When a character in a text string is preceded by an ampersand (&) and displayed in color using the DrawText function, the underscore used to indicate that the character is a keyboard mnemonic is drawn as a dotted line.

CAUSE
=====

If the text is drawn using any color other than COLOR_WINDOW, COLOR_WINDOWTEXT, or COLOR_GRAYTEXT, the underscore is drawn with the PatBlt function using a default gray brush. This results in a dotted underscore.

The TextOut function may be used in low-memory situations to draw text on a hard system-modal dialog box. Because of this, TextOut cannot create GDI objects or perform other allocations that may cause the Windows Kernel to move memory to satisfy the allocation.

The Windows version 3.0 USER module only caches brushes for the system colors COLOR_WINDOW, COLOR_WINDOWTEXT, and COLOR_GRAYTEXT. If the color used in a TextOut call does not match one of these colors, TextOut cannot create a new colored brush. The brush selected here is used to draw the underline on the mnemonic character.

RESOLUTION
==========

When drawing text with a mnemonic in color, an application must manually draw the underscore two pixels below the character. The GetCharWidth() function can be used to retrieve the exact width of the character.

STATUS
======

Microsoft has confirmed this to be a limitation of Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrPntDrawText

INF: How to Draw a Custom Window Caption
Article ID: Q99046

Summary:

Microsoft Windows draws captions for all eligible windows in the system. Applications need to specify only the WS_CAPTION style to take advantage of this facility. The current version of Microsoft Windows, however, imposes three significant restrictions on the captions. An application that does not want to be tied by any of these restrictions may want to draw its own caption. This article lists the restrictions and the steps required to draw a window caption.

It is important to note that future versions of Windows may not impose these restrictions. It is also important to note that an application should not draw its own caption unless it has very good reasons to do so. A window caption is a user interface object, and rendering it in ways different from other windows in the system may obstruct the user's conceptual grasp of the Microsoft Windows user interface.

More Information:

The three important restrictions imposed by Microsoft Windows version 3.1 on the caption for a window are:

- It consists of text only; graphics are not allowed.
- All text is centered and drawn with the system font.
- The length of the displayed caption is limited to 78 characters even when there is space on the caption bar to accommodate extra characters.

An application can essentially render its own caption consisting of any graphic and text with the standard graphics and text primitives by painting on the nonclient area of the window. The application should draw in response to the WM_NCPAINT and WM_NCACTIVATE messages. Trapping the WM_NCPAINT message without passing it on to the default window procedure [DefWindowProc()] is not recommended. An application should first pass on the WM_NCPAINT message to DefWindowProc() and then render its caption before returning from the message. This ensures that Microsoft Windows can properly draw the nonclient area. Because drawing the caption is part of DefWindowProc()'s nonclient area processing, an application should specify an empty window title to avoid any Windows-initiated drawing in the caption bar. The following steps indicate the computations needed to determine the caption drawing area:

1. Get the current window's rectangle using GetWindowRect(). This includes client plus nonclient areas and is in screen coordinates.

2. Get a device context (DC) to the window using GetWindowDC().

3. Compute the origin and dimensions of the caption bar. One needs to account for the window decorations (frame, border) and window bitmaps (min/max/system boxes). Remember that different window styles will result in different decorations and a different number of min/max/system boxes. Use GetSystemMetrics to compute the

dimensions of the frame, border, and the system bitmap dimensions.

4. Render the caption within the boundaries of the rectangle computed in step 3. Remember that the user can change the caption bar color any time by using the Control Panel. Some components of the caption, particularly text backgrounds, may need to be changed based on the current caption bar color. Use GetSysColor to determine the current color.

The following code sample draws a left-justified caption for a window (the code sample applies only to the case where the window is active):

Sample Code

```
        case WM_NCACTIVATE:
        if ((BOOL)wParam == FALSE)
          {
            DefWindowProc(hWnd, message, wParam, lParam);
            // Add code here to draw caption when widow is inactive.
              return TRUE;
        }
        // Fall through if wParam == TRUE, i.e., window is active.

        case WM_NCPAINT:
        // Let Windows do what it usually does. Let the window
        // caption be empty to avoid any Windows-initiated caption
        // bar drawing.
        DefWindowProc(hWnd, message, wParam, lParam);
        hDC = GetWindowDC(hWnd);
        GetWindowRect(hWnd, (LPRECT)&rc2);
        // Compute the caption bar's origin. This window has a system
        // box, a minimize box, a maximize box, and has a resizeable
        // frame.
        x = GetSystemMetrics(SM_CXSIZE) +
            GetSystemMetrics(SM_CXBORDER) +
            GetSystemMetrics(SM_CXFRAME);
        y = GetSystemMetrics(SM_CYFRAME);
        rc1.left = x;
        rc1.top = y;
        // 2*x gives twice the bitmap+border+frame size. Since
        // we only have two bitmaps, two borders, and one frame at
        // the end of the caption bar, subtract a frame to account
        // for this.

        rc1.right = rc2.right - rc2.left - 2*x -
                    GetSystemMetrics(SM_CXFRAME);
        rc1.bottom = GetSystemMetrics(SM_CYSIZE);
        // render the caption
        // Use the active caption color as the text background.
        SetBkColor(hDC, GetSysColor(COLOR_ACTIVECAPTION));
        DrawText(hDC, (LPSTR)"Left Justified Caption", -1,
                  (LPRECT)&rc1, DT_LEFT);
        ReleaseDC(hWnd, hDC);
        break;
```

Additional reference words: 3.00 3.10 minimum maximum

KbCategory: UsrPnt
KbSubCategory: UsrPntNcpaint

INF: GetClientRect() Coordinates Are Not Inclusive
Article ID: Q43596

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The coordinates returned by GetClientRect() are not inclusive. For example, to draw a border around the edge of the client area, draw it from the coordinates (Rectangle.left, Rectangle.top) to (Rectangle.right-1, Rectangle.bottom-1).

Addtional reference words: 3.0 3.1 3.00 3.10
KBCategory:
KBSubcategory: UsrPntRareMisc

INF: Using GetUpdateRgn()
Article ID: Q99047

Summary:

The documentation provided with the Microsoft Windows Software Development Kit (SDK) for Microsoft Windows version 3.1 does not address all the issues surrounding the use of the GetUpdateRgn() function. This article complements the documentation of GetUpdateRgn().

More Information:

The handle to the region (hRgn) does not have to be selected into a device context (DC) to be able to use it with GetUpdateRgn(). Even if the hRgn is selected as the clipping region for a DC, Windows only makes a copy of the hRgn for the hDC instead of using the hRgn directly.

When hRgn is used with GetUpdateRgn(), Windows will ignore any existing region in hRgn. It will replace any existing region with the current update region of the window.

Calling GetUpdateRgn() soon after BeginPaint() always yields an empty region because BeginPaint() validates the update region.

A typical Windows query function (functions that typically begin with Get and Is) does not initiate any action. GetUpdateRgn() is not a typical function in this respect. The third parameter, fErase, works as advertised to initiate a repaint on request.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcfategory: UsrPntInvalidat

BUG: ETO_CLIPPED Does Not Clip Rotated Text

Article ID: Q99110

Summary:

SYMPTOMS

In Microsoft Windows version 3.1, text output with a rotated TrueType font is not clipped when the ETO_CLIPPED flag and a clipping rectangle are specified in ExtTextOut().

CAUSE

This problem is due to a bug in Windows.

RESOLUTION

To workaround this problem, create a rectangular clipping region, select it into the device context, and do not specify the ETO_CLIPPED flag or pass in a clipping rectangle to ExtTextOut(). For example:

```
hRegion = CreateRectRgn(rc.left, rc.top, rc.right, rc.bottom);
SelectClipRgn(hDC, hRegion);
ExtTextOut(hDC, x, y, 0, NULL, szText, lstrlen(szText), NULL);
DeleteObject(hRegion);
```

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10 ExtTextOut clipping region TrueType DC
KBCategory:
KBSubcategory: UsrPntTextout

INF: SetResourceHandler() Return Value for Callback Function

Article ID: Q66951

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The callback function for SetResourceHandler() is documented on page 4-393 in the "Windows Software Development Kit Reference Volume 1" version 3.0 as having the following syntax:

```
   FARPROC FAR PASCAL LoadFunc(...)
```

The return value is incorrectly typed as being a FARPROC. It should be a HANDLE type and look like the following:

```
   HANDLE  FAR PASCAL LoadFunc(...)
```

The return value itself is not explained. The following is the explanation of the HANDLE type returned from the callback function:

```
   Return value: The return value is a HANDLE to a global piece of
                 memory that was allocated with GMEM_DDESHARE.
```

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrRscFindLoad

INF: FindResource() for Cursors and Menus

Article ID: Q49782

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

If you are using FindResource() on a Windows version 3.0 CURSOR or MENU, you must use RT_GROUP_CURSOR and RT_GROUP_MENU so that they can be found.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrRscFindLoad

INF: Save System Resources with One Control per Control Class

Article ID: Q80084

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

When an application is designed to gather a large amount of information from the user, the application can quickly consume many system resources due to the number of controls that it creates. Rather than creating many of the same type of control, an application can create one control and move it around.

The ONEEDIT sample in the Software/Data Library demonstrates this technique. ONEEDIT can be found in the Software/Data Library by searching on the word ONEEDIT, the Q number of this article, or S13260. ONEEDIT was archived using the PKware file-compression utility.

For more information on what system resources are and how they are consumed, query this knowledge base on the following words:

    prod(winsdk) and system and resources and heap and user and gdi

More Information:

ONEEDIT demonstrates collecting data for multiple fields with a single edit control. It collects the following data by moving one control:

    Name (First, Middle, Last)
    Sex
    Social Security Number
    Birthday (Month, Day, Year)
    Address
    City
    State
    Zip Code
    Phone

The results of using this technique are transparent to the user; however, an application designed in this way consumes far fewer system resources because one edit control is used rather than many edit controls. Fifteen fewer controls are required by ONEEDIT because it uses this technique. This savings becomes more significant as the number of controls within an application increases.

The ONEEDIT source includes an application programmer's interface (API) that can be added to an application. For more information on the API, see the file API.TXT included in the ONEEDIT archive file.

The remainder of this article lists some of the other techniques implemented in ONEEDIT.

1. Mouse support

   - The application performs hit-testing to move the edit control directly to any valid field.

2. Supports both TAB and SHIFT+TAB keys to move from one field to another

   - TAB: forward
   - SHIFT+TAB: backward

3. Input validation

   - If the value input for the State field does not correspond to a valid two-letter state abbreviation (as defined by the U. S. Postal Service), a Help dialog box is displayed listing the valid abbreviations.
   - Only a number (or a minus sign) is valid input in some fields (Social Security Number, Birthday, Zip Code, and Phone fields).
   - The value for the Sex field must be m, M, f, or F.

4. Special case processing to move between fields of the Phone number

   - Automatically advances to next phone field when one field is filled.
   - BACKSPACE moves to the preceding phone field (for example, from the beginning of the prefix to the end of the area code).
   - Arrow keys allow movement between fields.

5. Context-sensitive help in State field

   - The States Help can be invoked directly by pressing F2 when the cursor is in the State field.

6. Icon automatically changes

   - The icon displayed in ONEEDIT client area changes depending on the contents of the Sex field.

Additional reference words: limit control window user heap gdi 3.00
3.10
KBCategory:
KBSubcategory: UsrRsc

INF: Sample Code Extracts and Displays Application Resources
Article ID: Q81336

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

There are situations in which an application extracts information from another application's executable image file without loading the other application. The Microsoft Windows Program Manager does this when it extracts an icon from an EXE file to represent the application.

EXE (and DLL) files contain many resources and tables that may be useful in various circumstances. The EXEVIEW sample application demonstrates how to extract and decode these resources from application and library executable image files.

EXEVIEW can be found in the Software/Data Library by searching on the word EXEVIEW, the Q number of this article, or S13295. EXEVIEW was archived using the PKware file-compression utility.

Note: Because EXEVIEW uses code from the common dialog boxes dynamic-link library (COMMDLG.DLL), version 3.1 of the Microsoft Windows Software Development Kit (SDK) is required to build the sample. However, EXEVIEW will run under either Windows version 3.0 or 3.1 provided that the COMMDLG.DLL file is installed.

More Information:

EXEVIEW uses the information in both the Old Executable Header and the New Executable Header, each of which are documented in "The MS-DOS Encyclopedia" (Microsoft Press). EXEVIEW loads both headers and all the tables to which they refer. These tables include: the entry table, the segment table, the resource table, the resident and nonresident name tables, and the imported name table. EXEVIEW loads the resources listed in the resource table and displays them. Windows resources (icons, cursors, bitmaps, menus, and so forth) are displayed graphically. String tables and resource directories (of icons, cursors, fonts, and so forth) are listed in text format.

For more information on the file formats and resource formats, see the Windows SDK "Programmer's Reference, Volume 4: Resources," "The MS-DOS Encyclopedia," the September, 1991, issue of the "Microsoft Systems Journal," or the Microsoft Open Tools documentation.

Additional reference words: 3.10 softlib EXEVIEW.ZIP
KBCategory:
KBSubcategory: UsrRscFindLoad

INF: Accessing Resources from a Windows Executable File

Article ID: Q67708

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To get the ID values of resources from the header of a Windows executable (EXE) file, it is necessary to know about both the old and new EXE file header formats. For more information about the EXE file header formats, see the "MS-DOS Encyclopedia," published by Microsoft Press.

The data format for the individual resources in an EXE file is provided in a document on "Open Tools," which is currently available from Microsoft. For more information, query on the following words:

   Prod(winsdk) and Open and Tools and Relations

More Information:

The Software Library contains the sources for a sample program that demonstrates how to retrieve the ID value and offset for any resource in an executable or dynamic-link library (DLL) file. DUMPRES is a DOS (non-Windows) application that retrieves the data from the Windows application and displays it on the screen.

DUMPRES can be found in the Software/Data Library by searching on the keyword DUMPRES, the Q number of this article, or S12847. DUMPRES was archived using the PKware file-compression utility.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrRscFromExe

INF: Windows Resource Numbering Starts at 1

Article ID: Q11636

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Numbered resources in Windows must start at 1, not zero. Beginning with the MS-DOS version 4.0 executable format numbers segments from 1 and resources are implemented as independent segments. Therefore, resources are numbered from 1.

Additional reference words: 2.00 3.00
KBCategory:
KBSubCategory: UsrRscRareMisc

INF: Length of STRINGTABLE Resources

Article ID: Q20011

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In order to find the length of a string in the STRINGTABLE you need to do a FindResource() and then a SizeofResource() to find the total size in bytes of the current block of 16 strings. Remember that STRINGTABLEs are stored specially; to FindResource() you will ask for RT_STRING as the Type, and the (string number mod 16) + 1 as the name.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrRscRareMisc

INF: Defining Format Resources and Binding Them to EXEs

Article ID: Q41700

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The types of resources bound to a Windows application are not limited to those types defined in the Windows SDK; you can define your own type of resource. One way this can be done is described on Page 32 of the "Microsoft Windows Software Development Kit Programming Tools" manual Version 2.10. Using this technique, you can include information stored in another file in the executable.

There is an example in the Software Library of how to implement user-defined resources. This file can be found in the Software Library by searching on the filename MYRES.ARC, the Q number of this article, or S12223.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrRscUserDef

INF: Sample Code Stores Resources in a Dynamic-Link Library

Article ID: Q69029

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

DLLDLG2 is a file in the Software/Data Library that demonstrates using a dialog template stored in a Windows dynamic-link library (DLL).

DLLDLG2 can be found in the Software/Data Library by searching on the word DLLDLG2, the Q number of this article, or S12916. DLLDLG2 was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib DLLDLG2.ZIP
KBCategory:
KBSubcategory: UsrRscRareMisc

INF: Multiple References to the Same Resource

Article ID: Q83808

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Windows supports multiple references to a given resource. For example, suppose that an application has two top-level menus that each contain the same submenu. (An application can use the AppendMenu or SetMenu functions to add a submenu to another menu at run time.)

Normally, destroying a menu destroys all of its submenus. In the case above, however, when one menu is destroyed, the other menu has a lock on the common submenu. Therefore, the common submenu remains in memory and is not destroyed. The handle to the submenu remains valid until all references to the submenu are removed. The submenu either remains in memory or is discarded, while its handle remains valid.

More Information:

Windows maintains a lock count for each resource, including menus. When the lock count falls to zero, Windows can free (destroy) the object. Each time an application loads a resource, its lock count is incremented. If a resource is loaded more than once, only one copy is created; subsequent loads only increment the lock count. Each call to free a resource decrements its lock count.

When the LoadResource function determines if a resource has already been loaded, it also determines if the resource has been discarded. If so, LoadResource loads the resource again. The resource is not necessarily present in memory at all times. However, if the lock count is not zero and the resource is discarded, Windows will automatically reload the resource. All resources are discardable and will be discarded if required to free memory.

Therefore, in the example above, the application's call to the DestroyMenu function calls FreeResource, which checks the lock count. This process is analogous to LoadMenu, which calls LoadResource.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrRscFindLoad

INF: Do Not Call Destroy Function on Loaded Cursor or Icon

Article ID: Q84779

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" manual contains an error regarding the proper use of the DestroyCursor and DestroyIcon functions. This error is repeated a number of times throughout the manual. This article lists the various changes required to correct the error.

The manual states that when an application loads a cursor using the LoadCursor function, it should delete the cursor using the DestroyCursor function. Likewise, the manual states that when an application loads an icon using the LoadIcon function, it should delete the icon using the DestroyIcon function. These two statements are incorrect. DestroyCursor is used only on cursors created with the CreateCursor function and DestroyIcon is used only on icons created with the CreateIcon function.

To address this problem, make the following four changes to the "Microsoft Windows SDK: Programmer's Reference, Volume 2: Functions" manual:

- On page 216, modify the documentation for the DestroyCursor function to remove the reference to LoadCursor.

- On page 216, modify the documentation for the DestroyIcon function to remove the reference to LoadIcon.

- On page 576, modify the documentation for the LoadCursor function to replace the following line

  An application should use the DestroyCursor function to destroy any private cursors it loads.

  with the following line:

  An application should use the DestroyCursor function to destroy any private cursors it creates with the CreateCursor function.

- On page 577, modify the documentation for the LoadIcon function to replace the following line

  An application should use the DestroyIcon function to destroy any private icons it loads.

  with the following line:

An application should use the DestroyIcon function to destroy any private icons it creates using the CreateIcon function.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrRscFindLoad

INF: Sample Code to Extract an Icon from a Windows .EXE File
Article ID: Q70073

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

WINRES is a file in the Software/Data Library that provides the necessary structures to extract an icon from a Windows version 3.0 .EXE file. This information is obsolete in Windows version 3.1 since there is a new function named ExtractIcon in SHELL.DLL that performs this type of functionality.

WINRES can be found in the Software/Data Library by searching on the keyword WINRES, the Q number of this article, or S12937. WINRES was archived using the PKWARE file-compression utility.

More Information:

The following is a list of the files in the WINRES archive and how each is used:

Filename	Description
WINRES.C	Application to extract .ICO and .EXE resource information
WINOPEN.C	Used to display FileOpen dialog box and retrieve the filename
DUMPRES.H	.EXE header information derived from the "MS-DOS Encyclopedia" (Microsoft Press)

Program Theory of Operation

To retrieve the icons from an .ICO or .EXE file, the program must retrieve the device independent bitmaps (DIBs) that make up the icon. Although this is a difficult process, the necessary information is documented in the "MS-DOS Encyclopedia." A comment with each structure in the DUMPRES.H file provides a cross-reference to the relevant information in the "MS-DOS Encyclopedia." The .ICO file format is documented in Chapter 9, "File Formats," in version 3.0 of the "Microsoft Windows Software Development Kit Reference Volume 2."

Potential Enhancements to the Code
----

1. Choosing the Display Info menu item provides a listing of resource

   information about all resources in the file. This information
   should be placed into a list box, therefore the user will be able
   to scroll through the list. Currently, the information sometimes
   scrolls off the screen.

2. Pressing the I key on the keyboard will display the TEST.ICO file.
   Currently, this filename cannot be changed.

3. This version of the program does not support Windows 2.x .EXE
   files.

Differences Between .ICO Files and .EXE Resources

There are only three differences between an icon in an .ICO file and one in an .EXE file:

- Offset to the DIB

  In an .ICO file, the offset to the DIB is the last item in the icoResourceCount structure, icoDIBOffset. In an .EXE file, the offset is found in the offset for each individual icon, the rc_desc.wOffset field.

- The two reserved WORD fields before the icoDIBSize field in the .ICO file are filled in:

```
icoPlanes   -- Filled from the BITMAPINFOHEADER.biPlanes
icoBitCount -- Filled from the BITMAPINFOHEADER.biBitCount
```

  This allows an application to determine which icon is best suited for the display without having to read the DIB.

- The icoDIBOffset is changed from a DWORD in the .ICO file to a WORD in the .EXE file. The offset value in the .ICO file is changed to represent an ordinal number that ranges between 1 (one) and n (the total number of icons in the .EXE file). Note that one ICON group (value RT_GROUP_ICON), which is one icon rendered at two different resolutions, is counted as two separate icons (value RT_ICON).

Additional reference words: 3.00 softlib
KBCategory:
KBSubcategory: UsrRscFromExe

INF: Tracking Down Lost System Resources

Article ID: Q71455

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The term "system resources" refers to two scarce system-wide resources: the USER heap and the GDI heap. These two segments are each limited to 64K, and they are both shared by all the applications running under Windows.

During development of a Windows application, it is important to make sure that all the system resources allocated by the application at run time are released when the application terminates.

If too many system resources are lost, and either the USER heap or the GDI heap gets too full, performance will degrade for the entire Windows system.

The information below discusses how system resources can be "lost" by an application, and how to track down and correct such problems when they occur. In particular, it discusses:

1. Free System Resources -- A rough gauge of system resource use.

2. The GDI Heap -- The types of objects that are allocated in the GDI heap, and general rules for making sure that these objects are properly released.

3. The USER Heap -- The types of objects that are allocated in the USER heap, and general rules for making sure that these objects are properly released.

4. Troubleshooting and Heap Walker -- Specific techniques and tools for tracking down lost system resources.

More Information:

Free System Resources

The Program Manager's About Program Manager dialog box displays a Free System Resources value. This value indicates the amount of room left in the heap (USER or GDI) with the smallest amount of free space. There is no Windows API call that an application can use to obtain this value.

The easiest way to tell if an application is losing system resources is to examine Program Manager's Free System Resources value before and after

running the application. It is acceptable if this value goes down a little the first time the application is run, however if the value decreases every time the application runs and exits, system resources are being allocated and not released.

The GDI Heap

There are six GDI objects that a Windows program can create: pens, brushes, fonts, bitmaps, regions, and palettes. Space for each of these objects is allocated in the GDI heap.

Normally, the life cycle of a GDI object requires that the following steps be performed:

1. Create the GDI object.
2. Use the object.
3. Delete the object.

The DeleteObject call is used to delete most GDI objects.

The three general rules for deleting GDI objects are:

1. An application deletes all GDI objects that it creates.

2. Do not delete GDI objects while they are selected into a valid device context.

3. Do not delete stock objects.

Also, the following calls should always be matched:

```
CreateDC           -> DeleteDC
CreateCompatibleDC -> DeleteDC
CreateIC           -> DeleteDC
GetDC              -> ReleaseDC
BeginPaint         -> EndPaint
```

Creating GDI objects that are never destroyed is probably the most common cause of lost system resources. A careful examination of every place in the application's code that uses GDI objects will often reveal the problem.

The debugging version of Windows 3.1 will FatalExit when an application terminates if a GDI object owned by the application has not been deleted.

The USER Heap

When an application creates window classes, windows, and menus, these objects take up room in the USER heap. When an application terminates, Windows usually reclaims the memory used by objects in the USER heap.

Menus are a notable exception to this rule. Windows destroys the current menu of a window that is being destroyed. Windows does not destroy menus if they are not the current menu for any window. This can cause problems for applications that switch between multiple

menus: the "extra" menus are not automatically destroyed when the application terminates.

Therefore, if an application uses multiple menus, perform the following steps:

1. Keep the handle to each menu.

2. When the application is terminating, use the GetMenu function to discover which menu is currently being used. Do not call the DestroyMenu function on this menu, Windows will destroy it automatically.

3. Call the DestroyMenu function to destroy each of the other menus. This will reclaim the memory in the USER heap that these menus were using.

Because the USER heap is a shared system resource, it is important that an application does not allocate too many USER objects at once. If the USER heap becomes too full, subsequent calls to the RegisterClass and CreateWindow functions will fail.

This means that an application cannot create an excessively large number of windows. Also, when calling the RegisterClass functions, make sure that the cbWndExtra and cbClassExtra fields of the WNDCLASS structure are explicitly set to 0 (zero) if no extra bytes are needed.

Also, the following calls should always be matched:

```
CreateIcon   -> DestroyIcon
CreateCursor -> DestroyCursor
```

Troubleshooting and Heap Walker

As mentioned above, Program Manager's Free Systems Resources value can provide evidence that memory objects are not being reclaimed in the USER or GDI heaps.

It is sometimes unclear whether or not a particular Windows API call that creates an object must be balanced by a later call to explicitly delete the object. The following is a simple test that can be performed to find out:

1. Alter the Generic sample application to ensure that it contains a loop that makes the API call in question 50 times, creating 50 of the objects in question.

2. Run this version of Generic repeatedly. If the Program Manager's Free System Resources value goes down every time Generic runs and exits, a balancing API call must be made to reclaim the system resources before Generic terminates.

The most powerful tool for looking at problems with lost system resources is the Heap Walker application. Heap Walker is included with the Microsoft Windows Software Development Kit (SDK). Chapter 11 of the "Microsoft Windows Software Development Kit Tools" guide explains

how to use Heap Walker. In particular, page 11-8 outlines the procedure for checking for leftover GDI objects.

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrRsc

INF: WM_QUERYDRAGICON Return Value Documented Incorrectly

Article ID: Q72235

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

On page 6-94 of the "Microsoft Windows Software Development Kit Reference Volume 1," the documentation of the return value for the WM_QUERYDRAGICON message is incomplete. In addition to supplying the handle to a cursor in the low-order word of the return value, it is also possible to supply the handle to an icon in the low-order word of the return value. Windows will convert the icon to a black and white cursor and display the cursor while the user drags the icon.

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrRsc

PRB: Successful LoadResource of Metafile Yields Random Data

Article ID: Q86429

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

When an application for the Microsoft Windows graphical environment calls the LoadResource() function to load a metafile from the application's resources, locks the metafile with the LockResource() function, and uses the metafile, the application receives random data even though the LoadResource() and LockResource() functions indicate successful completion.

CAUSE

The application loaded the metafile previously and when the application freed the metafile, it used the DeleteMetaFile() function to invalidate the metafile handle.

RESOLUTION

Modify the code that unloads the metafile from memory to call the FreeResource() function.

More Information:

The LoadResource() and FreeResource() functions change the lock count for a memory block that contains the resource. If the application calls DeleteMetaFile(), Windows does not change the lock count. When the application subsequently calls LoadResource() for the metafile, Windows does not load the metafile because the lock count indicates that it remains in memory. However, the returned memory handle points to the random contents of that memory block.

For more information on the resource lock count, query in the Microsoft Knowledge Base on the following words:

multiple and references and LoadResource

Most of the time, an application uses the DeleteMetaFile() function to remove a metafile from memory. This function is appropriate for metafiles created with the CopyMetaFile() or CreateMetaFile() functions, or metafiles loaded from disk with the GetMetaFile() function. However, DeleteMetaFile() does not decrement the lock count of a metafile loaded as a resource.

Additional reference words: 3.00 3.10 EnumMetaFile GetMetaFile GetMetaFileBits PlayMetaFile PlayMetaFileRecord SetMetaFileBits SetMetaFileBitsBetter
KBCategory:
KBSubcategory: UsrRscFindLoad

INF: Working Around the STRINGTABLE 255 Character Limit

Article ID: Q74800

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Windows STRINGTABLE resources are limited to 255 characters per string. However, it is possible to work with longer strings using the technique described in this article.

More Information:

The following function, MyLoadString, will allow strings longer than 255 characters to be loaded. Using the exclamation mark (!) to indicate that a particular string is part of a longer whole is not required; any character that does not otherwise start a string may be used for this purpose. The string IDs used for pieces of the same string must be consecutive.

```
MyLoadString(HANDLE hInst, WORD wID, LPSTR szBuf)
{
  int i, j;
  char szLoadedString[256];   // temporary buffer

  *szBuf = 0;

  do
    {
    LoadString(hInst, wID, szLoadedString, 255);

    if ('!' == *szLoadedString)
      {
      lstrcat(szBuf, (szLoadedString + 1));
      wID++;
      }
    else
      lstrcat(szBuf, szLoadedString);
    }
  while ('!' == *szLoadedString);

  return lstrlen(szBuf);
}
```

The RC file will take on a form resembling the following:

```
STRINGTABLE
BEGIN
  100, "!This is the first part of the string to be loaded,"
  101, "!this is the second part, "
  102, "and this is the third"
END
```

Additional reference words: 3.0
KBCategory:
KBSubcategory: UsrRscRareMisc

INF: Placing Double Quotation Mark Symbol in a Resource String
Article ID: Q47674

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To specify a set of double quotation marks within a string in an application's resource (RC) file, use two double quotation mark characters in succession, as in the following example:

Specify the following string in the RC file:

```
   "The letter ""Q"" is quoted."
```

The following string will appear in the compiled resource (RES) file:

```
   The letter "Q" is quoted.
```

Additional reference words: 2.10 2.1 3.0 3.00
KBCategory:
KBSubcategory: UsrRscRareMisc

INF: STRINGTABLEs and Combined Resource Files

Article ID: Q51626

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

If you are developing a modular Windows application that includes multiple resource files, and each of the RC files contains several STRINGTABLE statements making your application quite large you may find that several of your "STRINGTALBE messages" are being omitted when you compile and run your application.

CAUSE

The identifier values for STRINGTABLE resources must be chosen with care. Multiple RC files may contain many STRINGTABLE statements, but the different RC files may not share the same ranges of identifier values. This is due to the method Windows uses to store and reference STRINGTABLE resources.

STRINGTABLE data is broken up into separate segments, each containing 16 strings. A modulo-16 algorithm is used to offset into a segment to select a specific string resource. The actual identifier values are used in constructing these segments. For example, consider the following RC file fragments:

```
/* Table 1 */          /* Table 2 */

STRINGTABLE            STRINGTABLE
BEGIN                  BEGIN
   1, "Hello"             3, "Error"
   2, "GoodBye"           4, "Memory"

END                    END
```

If Tables 1 and 2 are in a single RC file, the resulting RES file will contain a single STRINGTABLE segment containing Strings 1 through 4. The problem with this setup occurs when Tables 1 and 2 are compiled separately and combined with the DOS COPY command. Each original RES file will contain its own STRINGTABLE segment, but produces an overlapping set of segments when combined during the binary copy process. A LoadString() will fail to access the correct string since it looks in the wrong place for the requested string.

RESOLUTION

The solution is to use identifiers that do not share the same modulo-16 range over different RC files. To reflect this rule, the above example should look like the following:

```
/* Table 1 */          /* Table 2 */
```

```
STRINGTABLE            STRINGTABLE
BEGIN                  BEGIN
   1, "Hello"             17, "Error"
   2, "GoodBye"           18, "Memory"

END                    END
```

In this form, Tables 1 and 2 could be compiled separately and produce a correctly formed RES with the DOS COPY command.

Addtional reference words: 3.0
KBCategory:
KBSubcategory: UsrRscRareMisc

INF: SizeofResource() Rounds to Alignment Size
Article ID: Q57808

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SizeofResource() returns the resource size rounded up to the alignment size. Therefore, if you have your own resource types, you cannot use SizeofResource() to get the actual resource byte count.

It has been suggested that this be changed to reflect the actual number of bytes in the resource so that applications can use SizeofResource() to determine the size of each resource. This suggestion is under review and will be considered for inclusion in a future release of the Windows Software Development Kit (SDK).

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrRscUserDef

PRB: OpenFile Function Documented Incorrectly

Article ID: Q85328

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

The OpenFile function is documented on pages 4-322 through 4-325 of the "Microsoft Windows Software Development Kit Reference Volume 1" for Windows 3.0 and on 731 through 733 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" for Windows 3.1.

Each reference includes a table to document the third parameter of the OpenFile function. Both tables document the use of the OF_CANCEL and OF_PROMPT values incorrectly. The correct documentation is as follows:

Value	Meaning
OF_CANCEL	Does nothing.
OF_PROMPT	Displays a dialog box if the requested file does not exist. The dialog box informs the user that Windows cannot find the file. When the user chooses the Close button in the dialog box, OpenFile returns the HFILE_ERROR error value.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrSysRareMisc

INF: Possible Causes for System Resource Reduction

Article ID: Q66654

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

You can note via Program Manager's "About Box" that Windows' System Resources can be inadvertently reduced by a Windows application, causing subsequent RegisterWindow() or CreateWindow() function calls to fail.

The system resources are a reflection of the lower USER and GDI's data segments. The following steps eliminate the major causes of USER's data segment fill-up listed in each step:

1. Ensure that the cbClsExtraBytes and cbWndExtraBytes are set to 0 in the WNDCLASS structure, unless the application is definitely using them.

2. Ensure that a MakeProcInstance() is performed and the window procedure is EXPORTed in the .DEF file.

3. Use CVW to verify that the application is receiving the WM_CREATE message.

4. Confirm that all menus that are loaded or added are being destroyed.

The main reason why GDI's data segment fills up is that objects are being created and are not being destroyed when they are no longer needed. For more information, please refer to page 11-8 in the "Microsoft Windows Software Development Kit Tools" version 3.0 manual.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrSysRareMisc

INF: Sample Code Demonstrates SystemParametersInfo

Article ID: Q81140

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Version 3.1 of the Windows Software Development Kit documents a new function that allows an application to change most of the system settings. An application can retrieve and change settings as varied as the wallpaper on the desktop to the width of window borders using the SystemParametersInfo function.

SYSPARAM is a file in the Software/Data Library that demonstrates using the SystemParametersInfo function to retrieve and to change various parameters.

SYSPARAM can be found in the Software/Data Library by searching on the word SYSPARAM, the Q number of this article, or S13303. SYSPARAM was archived using the PKware file-compression utility.

Note: SYSPARAM modifies the WIN.INI file. Make a backup copy of WIN.INI before using this sample.

Additional reference words: 3.10 softlib SYSPARAM.ZIP
KBCategory:
KBSubcategory: UsrSysOtherCp

INF: GetCurrentTime() Rolls Over Every Hour
Article ID: Q20059

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The GetCurrentTime() function wraps (rolls over to zero) every hour.
In fact, it rolls after every 3,604,480 ticks.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrSysGetTime

FIX: System Metrics Not Updated by SwapMouseButton()

Article ID: Q69806

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

PROBLEM ID: WIN9103027

SYMPTOMS
   GetSystemMetics(SM_SWAPBUTTON) always returns the state of the
   mouse buttons as specified in the WIN.INI file.

CAUSE
   When the SwapMouseButton function is used to swap the function of
   the left and right mouse buttons, the system metric is not updated.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. This problem was corrected in Windows version 3.1.

More Information:

If it is necessary to keep track of the state of the mouse buttons, the application must call GetSystemMetrics(SM_SWAPBUTTON) during its initialization phase. This will return the state of the mouse buttons as indicated in the WIN.INI file. At each call to SwapMouseButton, the value of this Boolean variable should be toggled to reflect the current state of the mouse buttons.

If the change to the state of the mouse buttons is to be permanent, the WIN.INI file must be updated using the WriteProfileString function. Please note that the SwapMouseButton function is usually called only by the Control Panel. Although applications are free to call the function, the mouse is a shared resource. Reversing the state of the mouse buttons will affect all applications.

Additional reference words: 3.00 SR# G910202-11
KBCategory:
KBSubcategory: UsrSysGetMetric

INF: Sample Code to Provide Time and Date Information
Article ID: Q70070

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The information below describes a set of functions available in the Software/Data Library for formatting time and date strings using the international settings in the Windows Control Panel. This information applies to applications developed with the Windows Software Development Kit (SDK) version 3.0.

WINTIME is a file in the Software/Data Library that formats time and date strings according to the Control Panel settings. WINTIME can be found in the Software/Data Library by searching on the word WINTIME, the Q number of this article, or S12969. WINTIME was archived using the PKware file-compression utility.

More Information:

Windows version 3.0 allows the user to specify, with the Control Panel, the appropriate format for the date and time in the user's country. The Windows API provides no function to format strings of date and time information according to the Control Panel's settings. The sample code module WINTIME.C provides C source code for five functions that an application can use to format the date and time.

WINTIME.C is designed to be as flexible and portable as possible. The module can be used in an application or in a dynamic-link library (DLL) without modification.

Included with the WINTIME.C file is a sample application that demonstrates how to use the functions.

The five functions provided by WINTIME are documented below:

```
1. LPSTR FAR PASCAL TimeGetCurDate(LPSTR lpszDate, WORD wFlags)

   Description:
     TimeGetCurDate() returns the current date, formatted as specified
     by the user in the Control Panel.

   Arguments:
     Type/Name
              Description

     LPSTR lpszDate
              Points to a buffer that is to receive the date. Must be
              large enough to hold the longest date possible (64 bytes
              should be large enough).
```

```
   struct tm *lpTM
            Pointer to a "tm" structure (from TIME.H) that contains
            the time and date to be formatted.

   WORD wFlags
            Specifies flags that modify the behavior of the function.
            May be any of the following:

      Flag                Meaning
      ----                -------
      0                   The date will be formatted exactly the way
                          Control Panel does it, using Windows's
                          international settings. The date will be in
                          the long date format (Sunday, January 1,
                          1991).

      DATE_SHORTDATE      Forces the date to be formatted using the
                          short date format (1/01/91).

      DATE_NODAYOFWEEK    The date will be formatted without the day
                          of week. This flag may not be used with the
                          DATE_SHORTDATE flag (that is, January 1,
                          1991).

2. LPSTR FAR PASCAL TimeGetCurTime(LPSTR lpszTime, WORD wFlags)

   Description:
     TimeGetCurTime() returns the current time, formatted as specified
     by the user in the Control Panel.

   Arguments:
     Type/Name
              Description

     LPSTR lpszTime
              Points to a buffer that is to receive the formatted
              time. Must be large enough to hold the longest date
              possible (64 bytes should be large enough).

     WORD wFlags
              Specifies flags that modify the behavior of the function.
              May be any of the following:

        Flag                Meaning
        ----                -------
        0                   The time will be formatted exactly the way
                            Control Panel does it, using Windows's
                            international settings (for example,
                            1:00:00 AM).

        TIME_12HOUR         The time will be formatted in 12 hour format
                            regardless of the WIN.INI international
                            settings. This flag may not be used with the
                            TIME_24HOUR flag (for example, 2:32:10 PM).

        TIME_24HOUR         The time will be formatted in 24 hour format
                            regardless of the WIN.INI international
```

```
                     settings. This flag may not be used with the
                     TIME_12HOUR flag (for example, 14:32:10).

    TIME_NOSECONDS   The time will be formatted without seconds
                     (for example, 2:32 PM).


3. LPSTR FAR PASCAL TimeFormatDate(LPSTR lpszDate,
                                   struct tm FAR *lpTM,
                                   WORD wFlags)

   Description:
     TimeFormatDate() returns the date, specified in the tm structure,
     formatted as specified by the user in the Control Panel.

   Arguments:
     Type/Name
              Description

     LPSTR lpszDate
              Points to a buffer that is to receive the date. Must be
              large enough to hold the longest date possible (64 bytes
              should be large enough).

     struct tm *lpTM
              Pointer to a "tm" structure (from TIME.H) that contains
              the time and date to be formatted.

     WORD wFlags
              Specifies flags that modify the behavior of the function.
              May be any of the following:

        Flag               Meaning
        ----               -------
        0                  The date will be formatted exactly the way
                           Control Panel does it, using Windows's
                           international settings. The date will be in
                           the long date format (Sunday, January 1,
                           1991).

        DATE_SHORTDATE     Forces the date to be formatted using the
                           short date format (1/01/91).

        DATE_NODAYOFWEEK   The date will be formatted without the day
                           of week. This flag may not be used with the
                           DATE_SHORTDATE flag (for example, January 1,
                           1991).


4. LPSTR FAR PASCAL TimeFormatTime(LPSTR lpszTime,
                                   struct tm FAR *lpTM,
                                   WORD wFlags)

   Description:
     Function returns the time, specified in the tm structure,
     formatted as specified by the user in the Control Panel.
```

Arguments:

Type/Name
Description

LPSTR lpszTime
Points to a buffer that is to receive the formatted time. Must be large enough to hold the longest date possible (64 bytes should be large enough).

struct tm *lpTM
Pointer to a "tm" structure (from TIME.H) that contains the time and date to be formatted.

WORD wFlags
Specifies flags that modify the behavior of the function. May be any of the following:

Flag	Meaning
0	The time will be formatted exactly the way Control Panel does it, using Windows's international settings (for example, 1:00:00 AM).
TIME_12HOUR	The time will be formatted in 12 hour format regardless of the WIN.INI international settings. This flag may not be used with the TIME_24HOUR flag (for example, 2:32:10 PM).
TIME_24HOUR	The time will be formatted in 24 hour format regardless of the WIN.INI international settings. This flag may not be used with the TIME_12HOUR flag (for example, 14:32:10).
TIME_NOSECONDS	The time will be formatted without seconds (for example, 2:32 PM).

5. void FAR PASCAL TimeResetInternational(void)

Description:

This function sets some local static variables to Windows's current time and date format settings.

This function should be called every time a WM_WININICHANGE message is sent for the [intl] section and when the DLL or application using these functions is invoked.

These variables are in the DLL data segment. Therefore, if one application changes this information, the others will see the change.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrSysGetTime

INF: Calculating Free System Resources in Microsoft Windows

Article ID: Q61803

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows graphical environment, the About dialog box in the Program Manager displays a value that indicates the amount of free system resources available at a given time. The text below documents this calculation and the methods that an application can use to calculate this figure.

More Information:

Windows uses the same method to calculate the percentage of free system resources in each of its operating modes (real mode, standard mode, and enhanced mode under Windows 3.0, and standard mode and enhanced mode under Windows 3.1).

In the Windows environment, system resources are the amount of space available in the data segment (DS) for the user dynamic-link library (DLL) or the GDI DLL. Because system resources are exhausted if either of these data segments fills up, Windows reports the smaller of the two percentages for the system. The following formula describes the calculation for Windows 3.0 and 3.1; this may or may not be accurate in future versions of Windows:

```
   % free = min (% free of user DS, % free of GDI DS)

                (64K - current DS size) + free blocks in Heap
   % free DS = --------------------------------------------- * 100
                      64K - (size of statics and stack)
```

Or, in other words:

```
                 Current free heap assuming 64K
   % free DS =  -------------------------------- * 100
                   Maximum heap assuming 64K
```

Under Windows 3.1, an application can call the GetFreeSystemResouces function with the GFSR_SYSTEMRESOURCES value to obtain the percentage of free system resources. Because this function is not present in Windows 3.0, an application that runs in both environments must explicitly import this function from the user dynamic-link library (DLL) when the application runs under Windows 3.1.

Under Windows 3.0, an application can use the GetHeapSpaces function to determine the percentage of free system resources as follows:

```
   DWORD cbFree;
   WORD  wFreeK, wHeapK;
   WORD  wUserPercent, wGDIPercent, wSysPercent;

   cbFree = GetHeapSpaces(GetModuleHandle("USER"));
   wFreeK = LOWORD(cbFree) / 1024;
   wHeapK = HIWORD(cbFree) / 1024;
   wUserPercent = MulDiv(wFreeK, 100, wHeapK);

   cbFree = GetHeapSpaces(GetModuleHandle("GDI"));
   wFreeK = LOWORD(cbFree) / 1024;
   wHeapK = HIWORD(cbFree) / 1024;
   wGDIPercent = MulDiv(wFreeK, 100, wHeapK);

   wSysPercent = min(wUserPercent, wGDIPercent);
```

Because this technique does not provide accurate results under Windows 3.1, an application must determine the version of Windows with which it is running and perform the calculation accordingly.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrSysRareMisc

INF: Incomplete Description of ExitWindows()

Article ID: Q100359

Summary:

The description of the ExitWindows function does not list the code that can be used to terminate Windows and return control to MS-DOS.

More Information:

Page 290 of the Microsoft Windows Software Development Kit (SDK) "Programming Reference, Volume 2: Functions" manual for version 3.1 states that the ExitWindows function can be used to terminate Windows and return control to MS-DOS. However, the description doesn't list the code needed to exercise this feature of the ExitWindows function.

To allow the ExitWindows function to terminate Windows and return control to MS-DOS, pass a zero as the dwReturnCode parameter.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrSysRaremisc

INF: WM_TIMER Case on Page 94 of Guide to Programming Manual

Article ID: Q104789

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

Section 4.2.13, page 94, of the Windows SDK "Guide to Programming" manual shows how to add WM_TIMER processing to a window procedure. The code in the book is actually WM_LBUTTONDBLCLK. This is an error.

Chapter 4 of the "Guide to Programming" manual builds a sample program called input. The correct WM_TIMER case is found within INPUT.C, which is in the \GUIDE\INPUT directory of the Windows SDK directory. If this is a Visual C++ installation, INPUT.C will be in the \SAMPLES\INPUT directory.

The correct code is listed below:

```
   case WM_TIMER:
      wsprintf(TimerText, "WM_TIMER: %d seconds", nTimerCount += 5);
      InvalidateRect(hWnd, &rectTimer, TRUE);
      break;
```

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrTimWmTimer

INF: GetCurrentTime and GetTickCount Functions Identical

Article ID: Q45702

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

The GetCurrentTime and GetTickCount functions are identical. Each returns the number of milliseconds (+/- 55 milliseconds) since the user started Windows, and the return value of each function is declared to be a DWORD.

The following code demonstrates these two functions:

```
DWORD dwCurrTime;
DWORD dwTickCount;
char szCurrTime[50];

dwCurrTime = GetCurrentTime();
dwTickCount = GetTickCount();

sprintf(szCurrTime, "Current time = %lu\nTick count = %lu",
        dwCurrTime, dwTickCount);

MessageBox(hWnd, szCurrTime, "Times", MB_OK);
```

Additional reference words: 2.10 3.00 3.10 2.x SR# G890605-19712
KBCategory:
KBSubcategory: UsrTim

INF: Keeping a Window on Top of All Other Windows

Article ID: Q71573

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

By using a timer, a pop-up window can be made to behave like a "topmost" window, a window that remains on top of all other windows in the system. Such a window is brought to the top of all other windows whenever a WM_TIMER message is sent to its window function.

More Information:

To create a topmost window, first register a new window class. Using that new class, create a pop-up window without a parent. The window is created without a parent so that it is independent of the main overlapped window created for each application in the system. This will ensure that the topmost window remains open on the desktop even if the main window is minimized.

After creating this pop-up window, create a timer with the SetTimer() function and associate it with the window. Set the elapsed time to 100 milliseconds. The window function for the topmost window will process the WM_TIMER message and call SetWindowPos() to place the topmost window on top of all other windows. The code that processes the timer message may be similar to the following:

```
    case WM_TIMER:
        GetWindowRect(hWnd, &rect);

        SetWindowPos(hWnd,
                     NULL,
                     rect.left,
                     rect.top,
                     0,
                     0,
                     SWP_NOSIZE | SWP_NOMOVE | SWP_NOACTIVATE);
        break;
```

The window coordinates returned by GetWindowRect() are used in the call to SetWindowPos(). Doing this allows the user to move the topmost window (if it has a caption bar) and it will remain on top.

The wFlags parameter for SetWindowPos() also contains the SWP_NOACTIVATE flag. This is required so that the window can remain on top without moving the focus to that window. Failing to do this will disable all other applications in the system.

When the application that installed the topmost window is terminated, the timer should be removed using the KillTimer() function. This call

can be placed in the processing of the WM_DESTROY message in the topmost window's window procedure.

With this implementation, when a menu is activated, the screen will flash as the menu is painted and then the topmost window is painted over the menu. Menu selections can be made normally. However, depending upon the topmost window's position, not all menu items will be properly visible.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrTimWmtimer

INF: General Information About Windows WM_TIMER Messages

Article ID: Q44740

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Timers in the Windows environment are not synchronous; that is, when a timer "goes off," Windows does not stop processing and send a WM_TIMER message or call the timer event procedure. Instead, Windows sets some bits internally, to ensure that the task that "owns" the timer will run (as soon as the current task does something to yield control). When this task does eventually run and eventually calls GetMessage() or PeekMessage(), and there are no other messages available for this task, Windows will scan the timer list to see if any timers for this task have "gone off." If they have, Windows returns a WM_TIMER message or calls the timer event procedure. The following are two important facts that must be considered:

1. The WM_TIMER message and timer-event process ONLY happen when an application is sitting on a GetMessage() or PeekMessage(). They do not happen at timer interrupt time.

2. Only timers for the task that is currently active are scanned when looking for messages. A PARTICULAR TIMER IS OWNED BY THE TASK THAT WAS ACTIVE WHEN THE TIMER WAS CREATED.

The following is an example of the incorrect use of timers:

An application sets a keyboard hook. Inside the keyboard hook, a timer is set. It is important to realize that the keyboard hook is called from within GetMessage() or PeekMessage() whenever a task is pulling keys out of the system queue (the place where keyboard and mouse events go at interrupt level). The task that is active at the time the keyboard hook is called is completely random. Basically, it is the task that owns the windows with the focus.

Because a timer is owned by the active task at the time it is created, and any task may be the active task when the keyboard hook is called, the timer created in this manner will be owned by some random task. This, combined with the following facts, results in inconsistent, random behavior for this timer.

1. Timers are only scanned for the active task.

2. Timers are retrieved at message level.

3. All applications handle messages differently.

The basic rule is that a timer MUST NEVER be created from within a hook such as a keyboard hook.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrTimWmtimer

FIX: Incorrect Parameter Sent to SetTimer Callback Function

Article ID: Q75944

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

PROBLEM ID: WIN9109003

SYMPTOMS
   When an application calls the SetTimer function to create a timer
   that uses a callback function, and the callback function is called,
   the dwTime parameter to the callback function contains the callback
   function's procedure instance address. This parameter is documented
   to contain the current system time.

STATUS
   Microsoft has confirmed this to be a problem in Windows version
   3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrTimCallbacks

INF: How to Change a Window's Parent
Article ID: Q12172

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The following information describes how to change a window's parent in Windows versions 2.x and 3.x.

The correct way to do this in Windows versions 2.x is to destroy the child and then re-create the child with the proper handle to the parent. The following steps can be used to do this:

1. Do a ShowWindow(hWnd, HIDE_WINDOW) call.

2. Change the handle to the child using the SetWindowWord function.

3. Do a ShowWindow(hWnd, SHOW_WINDOW) call.

Note: Microsoft does not recommend that this method be used, and it is not guaranteed to work in future releases of Windows.

In Windows versions 3.x, the SetParent function should be used to set the parent of a window.

Additional reference words: TAR57085 2.00 2.03 2.10 2.x 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinGetwword

INF: Determining Visible Window Area When Windows Overlap
Article ID: Q75236

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There is no Windows API that reports the portion of an application's window that is not obscured by other windows. To determine which areas of the window are covered, it is necessary to walk through the window list managed by Windows.

Each window that precedes the application's window is "above" that window on the screen. Using the IntersectRect() function, check the rectangle of the window with any windows above to see if they intersect. Any window that is above the application's window and intersects its window rectangle obscures part of the application's window. By accumulating the positions of all windows that overlap the application's window, it is possible to determine which areas of the window are covered and which are not.

The following sample code demonstrates this procedure:

```
  GetWindowRect(hWnd, &rMyRect);    /*  Get the window dimensions
                                     *  for the current window.
                                     */
      /*  Start from the current window and use the GetWindow()
       *  function to move through the previous window handles.
       */
  for (hPrevWnd = hWnd;
      (hNextWnd = GetWindow(hPrevWnd, GW_HWNDPREV)) != NULL;
       hPrevWnd = hNextWnd)
    {
      /*  Get the window rectangle dimensions of the window that
       *  is higher Z-Order than the application's window.
       */
    GetWindowRect(hNextWnd, &rOtherRect);

      /*  Check to see if this window is visible and if intersects
       *  with the rectangle of the application's window. If it does,
       *  call MessageBeep(). This intersection is an area of this
       *  application's window that is not visible.
       */
    if (IsWindowVisible(hNextWnd) &&
           IntersectRect(&rDestRect, &rMyRect, &rOtherRect))
      {
      MessageBeep(0);
      }
    }
```

Additional reference words: 3.00

KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Process WM_GETMINMAXINFO to Constrain Window Size
Article ID: Q67166

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Microsoft Windows sends a WM_GETMINMAXINFO message to a window to determine the maximized size or position for the window, and the maximum or minimum tracking size for the window. An application can change these parameters by processing the WM_GETMINMAXINFO message.

Each window type has an absolute minimum size. If an application changes any of the values associated with WM_GETMINMAXINFO to a value smaller than the minimum, Windows will override the values specified by the application and use the minimum size. This minimum window size restriction has been removed from Windows version 3.1.

Note that Windows can send a WM_GETMINMAXINFO message to a window prior to sending a WM_CREATE message. Therefore, any processing for the WM_GETMINMAXINFO message must be independent of processing done for the WM_CREATE message.

More Information:

An application can use the WM_GETMINMAXINFO message to constrain the size of a window. For example, the application can prevent the user from changing a window's width while allowing the user to affect its height, or vice versa. The following code demonstrates fixing the width:

```
  int width;
  LPPOINT lppt;
  RECT rect;

  case WM_GETMINMAXINFO:
      lppt = (LPPOINT)lParam;   // lParam points to array of POINTs

      GetWindowRect(hWnd, &rect);   // Get current window size
      width = rect.right - rect.left + 1;

      lppt[3].x = width   // Set minimum width to current width
      lppt[4].x = width   // Set maximum width to current width

      return DefWindowProc(hWnd, message, wParam, lParam);
```

The modifications required to fix the height are quite straightforward.

For more information on the array of POINT structures that accompanies

the WM_GETMINMAXINFO message, please refer to the "Microsoft Windows Software Development Kit Reference."

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinMove

INF: Determining the Topmost Pop-Up Window

Article ID: Q66943

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When an application has many pop-up child windows (with a common parent window), the GetNextWindow() function can be used when one pop-up window is closed to determine the next topmost pop-up window that remains.

The following code fragment shows a window procedure for simple pop-up windows (modified from the PARTY program in Petzold's "Programming Windows"). In the WM_CLOSE case, the handle received by the pop-up window procedure is the handle of the pop-up to be closed. This sample activates the topmost pop-up window that remains by giving it the focus.

```
long FAR PASCAL PopupWndProc (hWnd, iMessage, wParam, lParam)
     HWND     hWnd;
     unsigned iMessage;
     WORD     wParam;
     LONG     lParam;
     {
     HWND     hWndPopup;

     switch (iMessage)
        {
        case WM_CLOSE:
            hWndPopup = GetNextWindow(hWnd, GW_HWNDNEXT);
            if (hWndPopup)
                SetFocus(hWndPopup);
            break;
        }

     return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
     }
```

Additional reference word: 3.00
KBCategory:
KBSubcategory: UsrWinZorder

INF: Ending the Windows Session from an Application

Article ID: Q75629

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The ExitWindows() function is provided to permit an application to close Windows.

In previous versions of Windows, an application sent the WM_ENDSESSION message to all windows to close the Windows session. Starting with version 3.0, ExitWindows() puts Windows into a special state so that Windows can perform housekeeping and unhook system interrupts to cleanly exit to DOS. Because an application cannot simulate this Windows special state, the application must call the ExitWindows() function to close Windows.

For more information on the ExitWindows() function, see page 4-130 of the "Microsoft Windows Software Development Kit Reference Volume 1."

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Reasons Why RegisterClass() and CreateWindow() Fail

Article ID: Q65257

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The RegisterClass() and CreateWindow() functions fail when the system resources are used up. The percentage of free system resources reflects the amount of available space in the USER and GDI heaps within Windows. The smaller amount of free space is reported in the Program Manager's About box because if either heap fills up, functions fail.

If the amount of free system resources remains low after the application is exited, it is more likely that the GDI heap is filling. The main reason for the GDI heap filling is that GDI objects that are created by the application are not deleted or destroyed when they are no longer needed, or when the program terminates. Windows does not delete GDI objects (pens, brushes, fonts, regions, and bitmaps) when the program exits. Objects must be properly deleted or destroyed.

The following are two situations that can cause the USER heap to get full:

1. Memory is allocated for "extra bytes" associated with window classes and windows themselves. Make sure that the cbClsExtra and cbWndExtra fields in the WNDCLASS structure are set to 0 (zero), unless they really are being used.

2. Menus are stored in the USER heap. If menus are added but are not destroyed when they are no longer needed, or when the application terminates, system resources will go down.

CreateWindow() will also fail under the following conditions:

1. Windows cannot find the window procedure listed in the CreateWindow() call. Avoid this by ensuring that each window procedure is listed in the EXPORTS section of the program's DEF file.

2. CreateWindow() cannot find the specified window class.

3. The hwndparent is incorrect (make sure to use debug Windows to see the RIPs).

4. CreateWindow() cannot allocate memory for internal structures in USER heap.

5. The application returns 0 (zero) to the WM_NCCREATE message.

6. The application returns -1 to the WM_CREATE message.

Additional reference words: 2.x 3.00
KBCategory:
KBSubcategory: UsrWinCreatewin

INF: Zooming Other Applications in Windows

Article ID: Q22425

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

An application can zoom, or maximize, another application by performing the following two steps:

1. Find the handle to the application's window by using the EnumWindows function.

2. Use the PostMessage function to post a message to that application's message queue as follows:

```
    if (fWindowToZoom)
       PostMessage(hWndToZoom, WM_SYSCOMMAND, SC_MAXIMIZE, 0L);
```

An application can use this code to maximize any application running under Windows.

Additional reference words: 2.x TAR60794
KBCategory:
KBSubcategory: UsrWinMove

INF: Placing a Status Bar in an MDI Frame Window

Article ID: Q65880

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Many applications that employ the Windows multiple document interface (MDI) provide information to the user through the use of a status bar at the bottom of the application. Microsoft Excel and Microsoft Word for Windows are two examples of applications that provide this context-sensitive assistance.

To create a status bar at the bottom of an MDI frame window, do the following:

1. Create an appropriately sized child of the frame window to display the status information.

2. Size the MDI client window so that it does not obscure the status window.

When the frame window changes size, it is necessary to size the MDI client and status windows so that the status bar will remain visible.

There is a sample application in the Software Library named MDISTAT that demonstrates how to implement a status bar within an MDI application. This file can be found in the Software/Data Library by searching on the word MDISTAT, the Q number of this article, or S12718. MDISTAT was archived using the PKware file-compression utility.

Additional reference words: 3.00 softlib MDISTAT.ZIP
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Subclassing the Desktop and Windows of Other Applications

Article ID: Q79276

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The code to subclass the desktop window, or windows that belong to other applications, must be placed in a dynamic-link library (DLL).

More Information:

In most cases, the desktop window uses the stack segment (SS) of the currently scheduled task. This is done for performance reasons because it reduces the number of task switches that take place when the desktop receives paint or mouse messages. In some cases, the desktop cannot be switched immediately to the currently scheduled task. In this situation, it will temporarily use the SS of the previously scheduled task.

When the desktop is subclassed, the data segment (DS) that the subclass procedure uses is specified by the hInstance used in a MakeProcInstance() call. However, the subclass procedure's SS depends on the currently scheduled task, as described earlier. Because the currently scheduled task may not be the task that subclassed the desktop, it is possible that SS != DS when the subclass procedure is executed. Therefore, the subclass procedure cannot assume that SS == DS. For this reason, the code to subclass the desktop must reside in a DLL where it is assumed that SS != DS.

When a window belonging to another application is subclassed, the DS that the subclass procedure uses is specified by the hInstance used in the MakeProcInstance() call. The SS used by the subclass procedure is the SS of the application that owns the window. Therefore, the subclass procedure cannot assume that SS == DS. For this reason, the code to subclass another application's window must reside in a DLL where it is assumed that SS != DS.

The DLL in which the subclass procedure is placed should be compiled with the Microsoft C Compiler's -Aw option, which tells the compiler that SS != DS. Using this option causes the compiler to generate a warning message when it detects the improper creation of a near pointer to an automatic variable.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrWinSubclass

INF: WindowFromPoint() Caveats

Article ID: Q65882

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When the coordinates passed to the WindowFromPoint() function correspond to a disabled, hidden, or transparent child window, the handle of that window's parent is returned.

To retrieve the handle of a disabled, hidden, or transparent child window, given a point, the ChildWindowFromPoint() function must be used.

More Information:

The following code fragment demonstrates the use of the ChildWindowFromPoint() function during the processing of a WM_MOUSEMOVE message. This code finds the topmost child window at a given point, regardless of the current state of the window.

In this fragment, hWnd is the window receiving this message and is assumed to have captured the mouse via the SetCapture() function.

```
HWND  hWndChild, hWndPoint;
POINT pt;
    .
    .
    .
    case WM_MOUSEMOVE:
        pt.x = LOWORD(lParam);
        pt.y = HIWORD(lParam);

        /*
         * Convert point to screen coordinates. When the mouse is
         * captured, mouse coordinates are given in the client
         * coordinates of the window with the capture.
         */
        ClientToScreen(hWnd, &pt);

        /*
         * Get the handle of the window at this point. If the window
         * is a control that is disabled, hidden, or transparent, then
         * the parent's handle is returned.
         */
        hWndPoint = WindowFromPoint(pt);

        if (hWndPoint == NULL)
            break;
```

```
        /*
         * To look at the child windows of hWnd, screen coordinates
         * need to be converted to client coordinates.
         */
        ScreenToClient (hWndPoint, &pt);

        /*
         * Search through all child windows at this point. This
         * will continue until no child windows remain.
         */
        while (TRUE)
            {
            hWndChild = ChildWindowFromPoint(hWndPoint, pt);

            if (hWndChild && hWndChild != hWndPoint)
                hWndPoint = hChild;
            else
                break;
            }

        // Do whatever processing is desired on hWndPoint

        break;
```

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinFrompoint

INF: Keeping a Window Iconic

Article ID: Q66244

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Normally, when an application's main window is being represented by an icon ("iconic"), you can restore it to an open window by double-clicking the icon or by choosing the Restore option from the System menu.

Opening the window can be prevented by placing code into the application that processes the WM_QUERYOPEN message by returning FALSE.

If it is necessary to perform processing before the iconic window is opened, the processing should be done in response to the WM_QUERYOPEN message. After processing is complete the program can return TRUE and the window will be opened.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinMove

PRB: WM_PARENTNOTIFY Message lParam Documented Incorrectly

Article ID: Q66248

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Page 6-93 of the "Microsoft Windows Software Development Kit Reference Volume 1" version 3.0 incorrectly documents the value of lParam sent with each WM_PARENTNOTIFY message.

The WM_PARENTNOTIFY message is sent to the parent of a child window when the child window is created, destroyed, or clicked with any mouse button. When the message is sent, wParam contains WM_CREATE, WM_DESTROY, or WM_*BUTTONDOWN for these situations, respectively. (Note that, in this context, * replaces one of L, M, and R, representing the left, middle, and right mouse buttons, respectively.)

The lParam parameter for this message is documented as follows:

```
   lParam    Contains the window handle of the child window in its
             low-order word and the ID of the child window in its
             high-order word.
```

This documentation is correct when wParam contains WM_CREATE or WM_DESTROY.

However, when wParam contains a WM_*BUTTONDOWN message, lParam contains the coordinates of the mouse click in the parent window's client coordinates. The low-order word of lParam contains the x-coordinate and the high-order word contains the y-coordinate.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinGeneral

INF: Obtaining an Application's Instance Handle

Article ID: Q35767

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Many Windows API functions require an application's instance handle as a parameter. This article describes three methods that an application can use to obtain its instance handle.

More Information:

Depending on the situation, an application might use any of the following three methods to obtain its instance handle:

1. The hInstance parameter to the WinMain procedure is the application's instance handle. The application can save this value in a global variable, making it available throughout the application.

2. The lParam parameter of the WM_CREATE message points to a CREATESTRUCT data structure, and the hInstance field of this structure is the application's instance handle. During the processing of the WM_CREATE message a window procedure can save this value in a static variable, as follows:

   ```
   hInst = ((LPCREATESTRUCT)lParam)->hInstance;
   ```

   This variable would then be available during all subsequent calls to the window procedure.

3. Given a window handle for one of its windows, an application can obtain its instance handle by calling the GetWindowWord function:

   ```
   GetWindowWord(hWnd, GWW_HINSTANCE)
   ```

Additional reference words: 2.03 2.10 2.x 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Window Handles of Global Objects

Article ID: Q10213

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A handle is a unique, fixed identifier for a global object. Handles exist to allow easy, indirect access to an object that may move in memory.

Handles are unique only among the objects to which they refer. A handle to a window will always be different from all other handles to windows, but it could have the same value as a handle to a brush (or to any other object).

It is safe to hold onto a window handle (or any handle) of a global object. As long as that object exists, its handle will not change.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinGeneral

PRB: DeferWindowPos Function Documented Incorrectly

Article ID: Q80324

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In version 3.0 of the "Microsoft Windows Software Development Kit Reference Volume 1," on page 4-81, the comments for the DeferWindowPos function incorrectly state the following:

> If the SWP_SHOWWINDOW or the SWP_HIDEWINDOW flags are set, scrolling and moving cannot be done simultaneously.

This statement is incorrect; if either of these two flags is set, the DeferWindowPos function will neither move nor size the windows. Use the ShowWindow function to show or hide windows.

In Windows version 3.1, these two flags can be set in the DeferWindowPos function to show or hide windows.

This documentation error does not exist in the Microsoft Windows Software Development Kit documentation for version 3.1.

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinMove

INF: Changing the Name of a Second Instance of an Application

Article ID: Q10856

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When running a Windows application with multiple instances, check hPrevInstance if you need to know if your instance is the first instance. If it is NULL, this is a new instance of static data; if it is not NULL, it is not a new instance of static data. You can give the instance a unique name by doing either of the following:

1. Use CreateWindow() and give it a unique name.

2. Use GetWindowLong() to access the CREATSTRUCT structure passed when processing a WM_CREATE message and specifically change the lpszName field (if you want to change the caption name on the existing, already named instance).

You cannot change the icon name dynamically. You should use the second method to update the icon name from untitled to a unique name.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinRareMisc

PRB: Return from EnableWindow() Documented Incorrectly

Article ID: Q66392

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

On Page 4-113 in the "Microsoft Windows Software Development Kit Reference Volume 1," version 3.0, the EnableWindow() function is incorrectly documented to return nonzero if the window is enabled or disabled as specified, or zero if the function fails.

Under Windows version 3.0, EnableWindow() returns TRUE if the window was previously disabled and FALSE if the window was previously enabled.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinEnabling

INF: Preventing Windows from Switching Tasks
Article ID: Q80822

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

Certain types of applications, such as security applications and possibly some setup programs, require the ability to disable task switching under Windows. To disable task switching, the program must prevent the user from accessing the Task List, using mouse or keystroke combinations to switch to other applications, and so forth.

The NOSWITCH application in the Software/Data Library demonstrates how an application can disable the task switch. NOSWITCH can be found in the Software/Data Library by searching on the word NOSWITCH, the Q number of this article, or S13281. NOSWITCH was archived using the PKware file-compression utility.

More Information:

NOSWITCH demonstrates how an application can disable task switching. The application must take the following four steps to disable task switching:

1. Subclass the desktop window to prevent the Task List from being displayed when the user double-clicks the desktop window.

2. Prevent the user from using key combinations to switch to another task by trapping the following three wParam values for the WM_SYSCOMMAND message: SC_NEXTWINDOW, SC_PREVWINDOW, and SC_TASKLIST.

3. Prevent other applications from activating themselves by enumerating all top-level windows, and disabling the windows that do not belong to this application.

4. Each time a WM_INITMEMUPOPUP message is received for the system menu, disable the menu items that relate to switching tasks.

Before exiting the application, reverse the effects of any actions taken to prevent switching tasks. Refer to the NOSWITCH sample for complete details.

There is a simpler, more limited, method to disable task switching. If the application is run as a maximized window, then step 2 above is enough to prevent switching tasks. However, the application must prevent the window from being restored or minimized. The limitation to this method is that the application must run as a maximized window.

Additional reference words: 3.00 3.10 3.x softlib NOSWITCH.ZIP
KBCategory:
KBSubcategory: UsrWinFocus

INF: Sample Code Simulates Changing List Box Style
Article ID: Q84978

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The windows styles specified when an application creates a list box are fixed throughout its lifetime. LBCHANGE is a file in the Software/Data Library that demonstrates using two list boxes to simulate changing the style of a list box.

LBCHANGE can be found in the Software/Data Library by searching on the word LBCHANGE, the Q number of this article, or S13439. LBCHANGE was archived using the PKware file-compression utility.

More Information:

When it creates a new window, Windows stores window style information in several internal data structures. Windows refers to these data structures when it performs window management tasks, including painting the window.

If an application calls the SetWindowLong function to change the window style dynamically, the new information is not reflected in the internal data structures. For example, an application can't effectively change a list box from the single selection style to the multiple selection style.

An application can work around this situation in two different ways:

- Create two list boxes with different styles. The application shows the list box with the desired style and hides the other.

  -or-

- Destroy the existing list box and create a new list box in its place.

The LBCHANGE sample demonstrates the first method, which is simple to implement and runs efficiently. However, this method requires memory space to store two copies of the list box contents.

Although the second method uses memory more efficiently, it requires more work to implement. If the application changes the list box style often, managing the contents of the list box by saving and reloading the contents and tracking the selected items can become quite involved.

Additional reference words: 3.00 3.10 listbox softlib LBCHANGE.ZIP

KBCategory:
KBSubcategory: UsrWinStyles

PRB: Exchanging Window Handles Between 2 Cooperating Programs
Article ID: Q11082

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

SYMPTOMS

There are two cooperating programs that have exchanged window handles. One of these programs is serving as a slave to the other, which is called the master. When a user closes the master program, it is necessary to close the slave program's window and terminate it.

If the window held by the slave is closed by the master program using DestroyWindow(), the slave program's window is destroyed; however, the slave program never receives a WM_DESTROY message.

RESOLUTION

Messages are sent to the task that created the window (that is, the master program). Note that the Windows message dispatcher has no means of "knowing" that a window handle has been exchanged.

For additional information query on the following words in this Knowledge Base:

    terminating and WM_CLOSE and top-level and DestroyWindow

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinGeneral

INF: Translating Client Coordinates to Screen Coordinates

Article ID: Q11570

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The GetClientRect function always returns the coordinates (0, 0) for the origin of a window. This behavior is documented in the "Microsoft Windows Software Development Kit (SDK) Programmer's Reference" manual.

To determine the screen coordinates for the client area of a window, call the ClientToScreen function to translate the client coordinates returned by GetClientRect into screen coordinates. The following code demonstrates how to use the two functions together:

```
RECT rMyRect;

GetClientRect(hwnd, (LPRECT)&rMyRect);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.left);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.right);
```

Additional reference words: Translate device 2.03 2.10 2.x 3.00 3.10 3.x

KBCategory:

KBSubcategory: UsrWinRect

INF: How to Create a Topmost Window
Article ID: Q81137

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Windows 3.1 introduces the concept of a topmost window that stays above all the non-topmost windows even when the window is not the active window.

There are two ways to add the topmost attribute to a window:

1. Use CreateWindowEx to create a new window. Specify WS_EX_TOPMOST as the value for the dwExStyle parameter.

2. Call SetWindowPos, specifying an existing non-topmost window and HWND_TOPMOST as the value for the hwndInsertAfter parameter.

SetWindowPos can also be used to remove the topmost attribute from a window. To do so, specify HWND_NOTOPMOST or HWND_BOTTOM as the value for the hwndInsertAfter parameter.

Additional reference words: 3.10
KBCategory:
KBSubcategory: UsrWinStyles

INF: Active Application, Active Window, Input Focus Definition

Article ID: Q35957

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Although the concepts of "input focus," "active application," and "active window" are very closely related, there are differences among them.

1. The "input focus" determines which window receives keyboard input.

2. The "active window" is the window that receives the user's attention. If it's an overlapped window or a pop-up window with a caption bar, the caption bar is highlighted. If it's a dialog window, the frame is highlighted. Either the active window or one of its child windows has the focus.

3. The "active application" is the application that created the window that has the input focus.

More Information:

The following discussion contains detailed definitions of the terms "input focus," "active window," and "active application," as well as a demonstration of those differences that you can perform using Windows Write.

The following definition is from the glossary of "Programmer's Guide to Windows, Second Edition," by David Durant, Geta Carlson, and Paul Yao, published by Sybex, page 647:

> Active Application
>
> The active application is the application that created the window that currently has the keyboard input focus. Applications do not need to be the active application in order to receive and process messages. Applications are notified by message whenever they are gaining or losing the status of "the active application." The user normally determines the active application, but applications can override this decision.

For more information on the active application, refer to the documentation for the WM_ACTIVATEAPP message in the "Microsoft Windows Software Development Kit (SDK) Reference, Volume 1."

The following is also from "Programmer's Guide to Windows, Second Edition," page 655:

> Focus

Keyboard input is transferred to the application as messages. Since several windows may be visible on the screen simultaneously, there must be a method for determining which of these windows should receive the keyboard input messages. In Windows, the window that has the focus is the window that will receive the keyboard messages. Normally, the user controls which window has the focus by use of the mouse, but applications can transfer the focus from window to window themselves.

For more information on focus, refer to the documentation for the WM_SETFOCUS and WM_KILLFOCUS messages and for the GetFocus, SetFocus, and EnableWindow functions in the "SDK Reference, Volume 1."

The following is from the book Programming Windows, Second Edition by Charles Petzold, published by Microsoft Press, pages 89-90:

Focus, Focus, Who's Got the Focus?

The keyboard must be shared by all applications running under Windows. Some applications may have more than one window, and the keyboard must be shared by these windows within the same application. When a key on the keyboard is pressed, only one window procedure can receive a message that the key has been pressed. The window that receives this keyboard message is the window with the "input focus."

The concept of input focus is closely related to the concept of "active window." The window with the input focus is either the active window or a child window of the active window. The active window is usually easy to identify. If the active window has a caption bar, Windows highlights the caption bar. If the active window has a dialog frame (a form most commonly seen in dialog boxes) instead of a caption bar, Windows highlights the frame. If the active window is an icon, Windows highlights the window's caption bar text below the icon.

The most common child windows are controls such as push buttons, radio buttons, check boxes, scroll bars, edit boxes, and list boxes that usually appear in a dialog box. Child windows are never themselves active windows. If a child window has the input focus, then the active window is its parent. Child window controls indicate that they have the input focus generally by using a flashing cursor or caret.

If the active window is an icon, then no window has the input focus. Windows continues to send keyboard messages to the icon, but these messages are in a different form from keyboard messages sent to active windows that are not icons.

A window procedure can determine when it has the input focus by trapping WM_SETFOCUS and WM_KILLFOCUS messages. WM_SETFOCUS indicates that the window is receiving the input focus, and WM_KILLFOCUS signals that the window is losing the input focus.

Petzold also indicates that Windows sends keyboard messages to icons differently than it does to windows with the focus. This difference is

noted in the following excerpt from the "Microsoft Windows Software Development Kit Reference, Volume 1," page 4-381:

> If a window is active but doesn't have the focus (that is, no window has the focus), any key pressed will produce the WM_SYSCHAR, WM_SYSKEYDOWN, or WM_SYSKEYUP message.

For more information on the active window, refer to the documentation for the WM_ACTIVATE message and for the GetActiveWindow and SetActiveWindow functions in the "Microsoft Windows Software Development Kit Reference, Volume 1."

Demonstration of Differences Between Focus, Active Application, and Active Window

The following is an experiment you can do that will clarify the differences among active application, active window, and input focus, and show you how the user can specify which attribute is assigned to which window:

Open Windows Write and choose Find from the Find menu. You'll now have a pop-up window that is a child of Write. Arrange your screen so that the Write, Find, and Program Manager windows are all on the screen together.

If you click in the Program Manager window, both the Write window's and the Find dialog window's caption bars will be set to the inactive color, and the Program Manager window's caption bar will be set to the active color. At this point, Program Manager is the active application; it is also the active window and has the input focus.

Click in the Write window; it will become the active application, as well as the active window, and will get the input focus.

Click in the Find window; although it is now the active window, it is not the active application (Write is) nor does it have the input focus: the focus is in the Find What box, which is an edit control window that is a child of the Find dialog window, which is itself a pop-up-style window that is a child of Write. (Note that in this discussion the term "child windows" is used in terms of parent-child relationships rather than the child window style WS_CHILD.)

At this point, you can move the input focus from box to box within the Find dialog window by using the TAB key (or by clicking with the mouse). There are three or four controls you can move between: Find What, Whole Word, Match Upper/Lowercase, and Find Next (available only if there is some text in the Find What box).

Now press ALT+F6. This will make Write the active window instead of Find, and the focus will be set to the Write window as well; this is indicated by the flashing vertical caret. Press ALT+F6 a second time. Find will be the active window again, and the input focus will be set to whichever control it was last set to (Find What, Whole Word, Match Upper/Lowercase, or Find Next).

Finally, you can use ALT+TAB to return to the Program Manager, making

it the active application, the active window, and the one with the input focus. If you press ALT+TAB again, you will make Write the active application, Find the active window, and set the input focus to one of Find's controls (whichever one last had the focus).

Additional reference words: 3.00 3.0 SR# G880919-3424
KBCategory:
KBSubcategory: UsrWinFocus

INF: Creating Applications that Task Manager Does Not Tile

Article ID: Q81708

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

A variety of applications that provide various types of information have been developed for the Windows graphical environment. Depending on the amount of information that the application presents, an application icon may provide enough screen space to show the information. The FREEMEM program in Chapter 5 of Charles Petzold's book "Programming Windows 3" (Microsoft Press) demonstrates presenting status data on an icon.

When an application has more data than can be readily represented on an icon, the user will often size the application to provide just the information necessary, and place the application's window in a convenient place on the screen. However, in this situation, if the user selects Cascade or Tile from the Task Manager window to arrange the active applications, the status window will probably change size and position on the screen.

This article presents a technique that an application can use to prevent itself from being cascaded or tiled by the Task Manager.

More Information:

To prevent an application from changing position when the Task Manager rearranges windows, perform the following four steps:

1. Create a minimum window procedure, called DummyWndProc, with the following code:

```
long FAR PASCAL DummyWndProc(HWND hwnd, WORD message,
                             WORD wParam, LONG lParam)
{
   return DefWindowProc(hwnd, message, wParam, lParam);
}
```

2. Register a minimum window class, DummyClass, that uses DummyWndProc as the class procedure. Specify the application's icon as the class icon.

3. Create an invisible pop-up window of class DummyClass. Specify the application's name for the window caption text. Optionally, the coordinates for this invisible window can specify a location off the screen.

4. In the CreateWindow call for the top-level visible window, specify the window from step 3 as the hwndParent parameter.

There is one caveat to using this method. If an application uses PostMessage with the hwnd parameter set to -1 to post a message to all applications, the invisible pop-up window will receive the message. If the application depends on any globally posted message, it is necessary to modify the DummyWndProc code above.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinMove

PRB: Documentation Errors for DeferWindowPos and SetWindowPos

Article ID: Q67676

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The "Microsoft Windows Software Development Kit Reference Volume 1" incorrectly documents the DeferWindowPos() and SetWindowPos() functions. The text explanation for the wFlags of these functions on pages 4-80 and 4-418, respectively, should be corrected to read as follows:

   WORD Specifies the 16-bit value that affects the size and position
   of the window. It can be any combination of values, listed below,
   created by using the bitwise OR operator.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinZorder

INF: Save and Restore Window Size, Position, Sample Code

Article ID: Q83234

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

Many applications available today allow the user to store the size and position of the application's main window in an initialization file. The next time the user runs the application, the stored size and position information is used to create the main window. The Windows File Manager and Program Manager are examples of applications that save and restore size and position information.

Under Windows 3.0, many operations are required to retrieve the normal size and position for a window if it is maximized or minimized. Under Windows 3.1, the GetWindowPlacement function has been provided to perform this task. GetWindowPlacement provides the size and position of a restored window at all times, even when the window is minimized or maximized. The SetWindowPlacement function allows an application to change its restored size and position at all times.

The GetWindowPlacement function stores the following information regarding a specified window into a WINDOWPLACEMENT structure:

 - Its current state (maximized, minimized or normal)
 - Its maximized position
 - Its minimized position
 - The rectangle that the window occupies in its normal position

The SetWindowPlacement function restores a specified window to the state described by a WINDOWPLACEMENT structure.

MDIREST is a file in the Software/Data Library that demonstrates using GetWindowPlacement to save the state of all windows in a multiple document interface (MDI) application when the application shuts down. When the user runs the application a second time, SetWindowPlacement uses the stored information to restore the state of the windows.

MDIREST can be found in the Software/Data Library by searching on the word MDIREST, the Q number of this article, or S13379. MDIREST was archived using the PKware file-compression utility.

Additional reference words: 3.00 3.10 softlib MDIREST.ZIP
KBCategory:
KBSubcategory: UsrWinMove

INF: Cases Where

Article ID: Q68583

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

In Windows version 3.0, the "normal" size and position of a window is not available when that window is maximized (zoomed) or minimized (represented as an icon). In Windows version 3.1, two new functions named GeWindowPlacement and SetWindowPlacement have been added which provide access to normal position information.

The remainder of this article provides two possible ways to work around this limitation in Windows version 3.0:

1. If the normal size is needed only as the application is shut down, restore the window and retrieve its position before closing the application. The following call can be used to restore the window whose window handle is hWnd:

   ```
   SendMessage(hWnd, WM_SYSCOMMAND, SC_RESTORE, 0L);
   ```

   The GetWindowRect or GetClientRect functions can then be used to obtain the window's position.

2. If the normal size is needed at all times, keep track of the position every time the window receives a WM_MOVE message. If the IsIconic and IsZoomed functions both return FALSE, assume the window is normal and update the position values. Otherwise, do not change the saved position information.

Additional reference words: 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinMinmax

INF: Using the GetWindow() Function

Article ID: Q33161

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

An application can use the GetWindow function to enumerate the windows that have a specified relationship to another window. For example, an application can determine the windows that are children of the application's main window.

The GetWindow function returns NULL when no more windows match the specified criteria. Given a window handle, hWnd, the following code determines how many siblings the associated window has:

```
int CountSiblings(HWND hWnd)
   {
   HWND hWndNext;
   short nCount = 0;

   hWndNext = GetWindow(hWnd, GW_HWNDFIRST);
   while (hWndNext != NULL)
      {
      nCount++;
      hWndNext = GetWindow(hWndNext, GW_HWNDNEXT);
      }

   return nCount;
   }
```

Additional reference words: 2.00 2.03 2.10 3.00 3.10 2.x
KBCategory:
KBSubcategory: UsrWinEnumerate

INF: Sample Code Demonstrates Windows 3.1 Window Styles
Article ID: Q83556

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Microsoft Windows version 3.1 introduces five new window styles and two new extended window styles. An application can specify any of the five new window styles as the dwStyle parameter for the CreateWindow function. Likewise, an application can specify either of the two new extended window styles as the dwExStyle parameter for the CreateWindowEx function.

STYLE is a file in the Software/Data Library that demonstrates using all the styles listed below except for MDIS_ALLCHILDSTYLES. STYLE can be found in the Software/Data Library by searching on the word STYLE, the Q number of this article, or S13402. STYLE was archived using the PKware file-compression utility.

More Information:

The text below explains the behavior of windows created with the window styles added to Windows 3.1.

Combo Box Style

CBS_DISABLENOSCROLL -- The list box portion of a combo box shows a disabled vertical scroll bar when the list box can display all items it contains without scrolling the contents. Without this style, the scroll bar is hidden when the list box can display all items.

Edit Control Styles

ES_READONLY -- The user cannot enter or edit text in the edit control.

ES_WANTRETURN -- This style can be used for a multiline edit control in a dialog box. Each time the user presses the ENTER key while the multiline edit control has the focus, Windows adds a carriage return to the edit control's contents. Without this style, pressing the ENTER key triggers the dialog box's default push button. This style does not affect a single-line edit control.

List Box Style

LBS_DISABLENOSCROLL -- A list box shows a disabled vertical scroll bar when the list box can display all items it contains. Without this style, the scroll bar is hidden when the list box can display all

items.

MDI Client Style

MDIS_ALLCHILDSTYLES -- This style can be specified when an application creates multiple document interface (MDI) client windows. When an application creates MDI children of an MDI client window that has the MDIS_ALLCHILDSTYLES style, Windows will create the children with exactly the styles specified in the style field of the MDICREATESTRUCT structure; the default MDI child window styles are not enforced.

The text below explains the behavior of windows created with the extended window styles added to Windows 3.1.

WS_EX_ACCEPTFILES -- The window will accept files using the drag-drop interface.

WS_EX_TOPMOST -- Windows places the window above all non-topmost windows. The window will stay above all non-topmost windows even when the window is deactivated. After a window is created, an application can use the SetWindowPos function to add or remove this attribute.

Additional reference words: 3.10 softlib STYLE.ZIP
KBCategory:
KBSubcategory: UsrWinStyles

INF: Adding and Removing Caption of a Window
Article ID: Q83915

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

NOTITLE is a file in the Software/Data Library that demonstrates adding and removing the title (caption) of a window.

NOTITLE can be found in the Software/Data Library by searching on the word NOTITLE, the Q number of this article, or S13405. NOTITLE was archived using the PKware file-compression utility.

More Information:

In the NOTITLE sample, when the window has a caption and the user selects NoTitle from the main menu, the caption is removed. When the user double-clicks the window without a caption by using the left mouse button, the caption is added to the window.

NOTITLE removes the caption by performing the following five steps:

1. Removes the menu from the window by calling the SetMenu function with NULL as the hMenu parameter.

2. Retrieves the current window style by calling GetWindowLong with GWL_STYLE as the nIndex parameter.

3. Removes the caption bit.

4. Calls SetWindowLong with GWL_STYLE to change the style of the window.

5. Calls InvalidateRect to instruct the application to redraw the window.

NOTITLE places the caption on the window by performing the following five steps:

1. Adds the menu to the window by calling the SetMenu function with the menu handle as the hMenu parameter.

2. Retrieves the current window style by calling GetWindowLong with GWL_STYLE as the nIndex parameter.

3. Adds the caption bit.

4. Calls SetWindowLong with GWL_STYLE to change the style of the window.

5. Calls InvalidateRect to instruct the application to redraw the
   window.

Additional reference words: 3.00 3.10 softlib NOTITLE.ZIP
KBCategory:
KBSubcategory: UsrWinStyles

PRB: Window Dragged Close to Screen Edge Becomes Invisible

Article ID: Q31075

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

If the user uses the mouse to drag the left edge of a window (or dialog box) within a few pixels of the right edge of the display and releases the mouse button, the window moves completely off the display. The user can't use the mouse to bring the window back into the display.

CAUSE

The window is created with the CS_BYTEALIGNCLIENT or CS_BYTEALIGNWINDOW class style bit set. When the user releases the mouse button, the window moves right to byte-align itself. However, because the byte boundary is off the screen, the entire window becomes invisible.

RESOLUTION

There are two methods to work around this problem:

- Give the window a system menu. This will prevent it from disappearing off the right edge of the screen.

  -or-

- Calculate the size of any window that has a caption bar to conform to the following formula:

```
horizontal_size modulo 8 <= 4
```

Additional reference words: 2.00 2.03 2.10 2.x 3.00 3.10 TAR77457 move
KBCategory:
KBSubcategory: UsrWinMove

INF: Window Owners and Parents

Article ID: Q84190

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Windows environment, two relationships that can exist between windows are the owner-owned relationship and the parent-child relationship.

The owner-owned relationship determines which other windows are automatically destroyed when a window is destroyed. When window A is destroyed, Windows automatically destroys all of the windows owned by A.

The parent-child relationship determines where a window can be drawn on the screen. A child window (that is, a window with a parent) is confined to its parent window's client area.

This article discusses these relationships and some Windows functions that provide owner and parent information for a given window.

More Information:

A window created with the WS_OVERLAPPED style has no owner and no parent. Alternatively, the desktop window can be considered the owner and parent of a WS_OVERLAPPED-style window. A WS_OVERLAPPED-style window can be drawn anywhere on the screen and Windows will destroy any existing WS_OVERLAPPED-style windows when it shuts down.

A window created with the WS_POPUP style does not have a parent by default; a WS_POPUP-style window can be drawn anywhere on the screen. A WS_POPUP-style window will have a parent only if it is given one explicitly through a call to the SetParent function.

The owner of a WS_POPUP-style window is set according to the hWndParent parameter specified in the call to CreateWindow that created the pop-up window. If hWndParent specifies a nonchild window, the hWndParent window becomes the owner of the new pop-up window. Otherwise, the first nonchild ancestor of hWndParent becomes the owner of the new pop-up window. When the owner window is destroyed, Windows automatically destroys the pop up. Note that modal dialog boxes work slightly differently. If hWndParent is a child window, then the owner window is the first nonchild ancestor that does not have an owner (its top-level ancestor).

A window created with the WS_CHILD style does not have an explicit owner; it is implicitly owned by its parent. A child window's parent is the window specified as the hWndParent parameter in the call to CreateWindow that created the child. A child window can be drawn only

within its parent's client area, and is destroyed along with its parent.

An application can change a window's parent by calling the SetParent function after the window is created. Windows does not provide a method to change a window's owner.

Windows provides three functions that can be used to find a window's owner or parent:

- GetWindow(hWnd, GW_OWNER)
- GetParent(hWnd)
- GetWindowWord(hWnd, GWW_HWNDPARENT)

GetWindow(hWnd, GW_OWNER) always returns a window's owner. For child windows, this function call returns NULL. Because the parent of the child window behaves similar to its owner, Windows does not maintain owner information for child windows.

The return value from the GetParent function is more confusing. GetParent returns zero for overlapped windows (windows with neither the WS_CHILD nor the WS_POPUP style). For windows with the WS_CHILD style, GetParent returns the parent window. For windows with the WS_POPUP style, GetParent returns the owner window.

GetWindowWord(hWnd, GWW_HWNDPARENT) returns the window's parent, if it has one; otherwise, it returns the window's owner.

Two examples of how Windows uses different windows for the parent and the owner to good effect are the list boxes in drop-down combo boxes and the title windows for iconic MDI (multiple document interface) child windows.

Due to its size, the list box component of a drop-down combo box may need to extend beyond the client area of the combo box's parent window. Windows creates the list box as a child of the desktop window (hWndParent is NULL); therefore, the list box will be clipped only by the size of the screen. The list box is owned by the first nonchild ancestor of the combo box. When that ancestor is destroyed, the list box is automatically destroyed as well.

When an MDI child window is minimized, Windows creates two windows: an icon and the icon title. The parent of the icon title window is set to the MDI client window, which confines the icon title to the MDI client area. The owner of the icon title is set to the MDI child window. Therefore, the icon title is automatically destroyed with the MDI child window.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrWinGeneral

INF: Allocating and Using Class and Window Extra Bytes
Article ID: Q34611

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The WNDCLASS structure contains two fields, cbClsExtra and cbWndExtra, which can be used to specify a number of additional bytes of memory to be allocated to the class structure itself or to each window created using that class.

Every application that uses class extra bytes and window extra bytes must specify the appropriate number of bytes before the window class is registered. If no bytes are specified, an attempt to store information in extra bytes will cause the application to write into some random portion of Windows memory, causing data corruption.

Windows version 3.1 will FatalExit if extra bytes are used improperly.

If an application does not use class extra bytes or window extra bytes, it is important that the cbClsExtra and cbWndExtra fields be set to zero.

Class and window extra bytes are a scarce resource. If more than 4 extra bytes are required, use the GlobalAlloc fucntion to allocate a block of memory and store the handle in class or window extra bytes.

Class Extra Bytes

For example, setting the value of the cbClsExtra field to 4 will cause 4 extra bytes to be added to the end of the class structure when the class is registered. This memory is accessible by all windows of that class. The number of additional bytes allocated to a window's class can be retrieved through the following call to the GetClassWord function:

```
   nClassExtraBytes = GetClassWord(hWnd, GCW_CBCLSEXTRA);
```

The additional memory can be accessed one word at a time by specifying an offset, in BYTES (starting at 0), as the nIndex parameter in calls to the GetClassWord function. These values can be set using the SetClassWord function.

The GetClassLong and SetClassLong functions perform in a similar manner and get or set four bytes of memory respectively.

It is important to note that while the functions get or set a word of information, the offset is a number of bytes.

Window Extra Bytes

Assigning a value to cbWndExtra will cause additional memory to be allocated for each window of the class. If, for example, cbWndExtra is set to 4, every window created using that class will have 4 extra bytes allocated for it. This memory is accessible only by using the GetWindowWord and GetWindowLong functions, and specifying a handle to the window. These values can be set by calling the SetClassWord or SetClassLong functions. As with the class structures, the offset is always specified in bytes.

An example of using window extra bytes would be a text editor that has a variable number of files open at once. The file handle and other file-specific variables can be stored in the window extra bytes of the corresponding text window. This eliminates the requirement to always consume memory for the maximum number of handles or to search a data structure each time a window is opened or closed.

Additional reference words: 2.10 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinProps

INF: Dangers of Uninitialized Data Structures

Article ID: Q74277

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In general, all fields in structures passed to functions in the Microsoft Windows graphical environment should be initialized. If a field is not initialized, it may contain random data, which can cause unexpected behavior.

For example, before an application registers a window class, it must initialize the cbClsExtra and cbWndExtra fields of the WNDCLASS data structure. Windows allocates cbClsExtra bytes for the class, and cbWndExtra bytes for each window created using the class. If these fields contain large random values, the application may run out of memory quickly.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Using SetClassLong Function to Subclass a Window Class

Article ID: Q32519

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

It is possible to subclass an entire window class by using the SetClassLong function. However, doing so will only subclass windows of that class created after the call to the SetClassLong function. Windows created before the call to the SetClassLong function are not affected.

More Information:

Calling the SetClassLong function with the GCL_WNDPROC index changes the class function address for that window class, creating a subclass of the window class. When a subsequent window of that class is created, the new class function address is inserted into its window structure, subclassing the new window. Windows created before the call to the SetClassLong function (in other words, before the class function address was changed) are not subclassed.

An application should not use the SetClassLong function to subclass standard Windows controls such as edit controls or buttons. If, for example, an application were to subclass the entire "edit" class, then subsequent edit controls created by other applications would be subclassed.

An application can subclass individual standard Windows controls that it has created by calling the SetWindowLong function.

Additional reference words: edit static button listbox scrollbar 2.03 2.10 2.x 3.00 3.10
KBCategory:
KBSubcategory: UsrWinGetwword

PRB: DEF File Exports Statement Documentation Error

Article ID: Q69892

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SYMPTOMS

The documentation for the EXPORTS statement in the module-definition file on pages 10-4 and 10-5 of the "Microsoft Windows Software Development Kit Reference Volume 2" for versions 3.0 and 3.1 is incomplete.

RESOLUTION

The name of any function that is called using the C calling convention must be preceded with an underscore (_) when it is listed in the EXPORTS section.

For example, listed below is the declaration for the MyVarArgs function:

```
int FAR CDECL MyVarArgs(LPSTR lpArgList, ...)
```

In the DEF file, the MyVarArgs function would be exported with the following statement:

```
EXPORTS
  _MyVarArgs
```

Additional reference words: docerr 3.00 3.10 3.x MICS3 R10
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: WinQuickSort(), qsort() Replacement for Windows Available
Article ID: Q70006

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Software Library contains object modules that implement the Quicksort similar to the C run-time library function qsort(). These object modules are compatible with Windows dynamic-link libraries (DLLs) and work with FAR data pointers, including memory obtained through GlobalAlloc().

There are two versions of the object module, WINQSRTS.OBJ and WINQSRTM.OBJ, for small-model and medium-model applications, respectively. The small-model version uses a near pointer to the comparison function, whereas the medium-model version uses a far pointer.

The object modules and a test program can be found in the Software/Data Library by searching on the keyword WINQSORT, the Q number of this article, or S12939. WINQSORT was archived using the PKWARE file-compression utility.

Additional reference words: 3.0 3.00
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Reactivating First (and Only) Instance of an Application

Article ID: Q70074

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

Sometimes restricting an application to a single instance is desirable. In that case, when the user starts a second instance of the application, a message box that says that the application is already running can be displayed. A more user-friendly approach is to bring the first instance to the front and activate it.

To accomplish this, perform the following four steps:

1. Obtain the main window handle of the first instance.

2. If the first instance has an open dialog box, obtain the window handle of its last active pop-up window.

3. Bring the parent and pop-up window (if present) to the front.

4. Do not run the second instance.

More Information:

To determine whether an instance of the application is already running, the first method below uses the FindWindow function to look for the an application with the same main window class name. The second method uses the hPrevInstance parameter to the WinMain function.

The FindWindow method does not depend on any memory architecture or on the hPrevInstance parameter; therefore, it is the suggested method for future portability. The hPrevInstance assumes a memory architecture for the operating system that allows a task to access another application's data segments. In future versions of Windows, this may not be possible if applications do not share the same local descriptor table (LDT).

Method 1: Use FindWindow Function

```
HWND FirsthWnd, FirstChildhWnd;

if (FirsthWnd = FindWindow("MyMainWindowClassName", NULL))
   {
   // Found another running application with the same class name.
   // Therefore, one instance is already running.
   // Note: "MyMainWindowClassName" must match the class name of the
```

```
   // program's main window. In the GENERIC sample application,
   // provided with the Microsoft Windows Software Development Kit,
   // versions 3.0 and 3.1, the chass name is "GenericWClass", which
   // is used in the InitApplication and InitInstance functions.

   FirstChildhWnd = GetLastActivePopup(FirsthWnd);
   BringWindowToTop(FirsthWnd);         // bring main window to top

   if (FirsthWnd != FirstChildhWnd)
      BringWindowToTop(FirstChildhWnd); // a pop-up window is active
                                        // bring it to the top too

   return FALSE;                        // do not run second instance
   }
```

Method 2: Use hPrevInstance Parameter

```
// Declare a global variable to save the handle of the first instance
// of the main window.

HWND FirsthWnd;


// Make the following modifications to the WinMain function:

HWND FirstChildhWnd;    // handle of last active pop-up window of the
                        // first application instance

if (!hPrevInstance)     // other instances of application running?
   if (!InitApplication(hInstance))  // initialize shared things
         return FALSE;  // exits if unable to initialize
   else {}

else  // a previous instance exists;
      // retrieve the main window handle from the first instance
   {
   GetInstanceData(hPrevInstance, (NPSTR)&FirsthWnd, 2);

   FirstChildhWnd = GetLastActivePopup(FirsthWnd);
   BringWindowToTop(FirsthWnd);         // bring main window to top

   if (FirsthWnd != FirstChildhWnd)
      BringWindowToTop(FirstChildhWnd); // a pop-up window is active
                                        // bring it to the top too

   return FALSE;                        // do not run second instance
   }


// Add this line to the InitApplication function:

FirsthWnd = NULL;  // no previous window, so this is NULL


// Add these two lines to the InitInstance function after the
```

```
// "hWnd = CreateWindow(...);" call:

if (!FirsthWnd)       // If this is the first instance,
   FirsthWnd = hWnd;  // save the window handle.
```

Additional reference words: 3.00 3.10 SR# G901025-65
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Sample Application Splits a Window into Two Panes
Article ID: Q85633

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

SPLITWIN is a file in the Software/Data Library that demonstrates splitting a window into two panes using the mouse or keyboard. Each pane can be scrolled independently in the vertical direction. Scrolling the lower child horizontally, however, causes the upper child to scroll by the same amount.

SPLITWIN can be found in the Software/Data Library by searching on the word SPLITWIN, the Q number of this article, or S13473. SPLITWIN was archived using the PKware file-compression utility.

More Information:

The SPLITWIN sample creates a frame window that has a child window. The child window fills the client area of the frame window. When the user splits the window, SPLITWIN creates a second child window. The two child windows are sized according to the position of the split bar.

The user can change the position of the split bar using either the mouse or the arrow keys. After the window is split, it can be restored to its original state by dragging the split bar off the frame window or by selecting a position that does not provide enough room to create the second window.

When the user clicks the mouse button in the split bar, SPLITWIN starts a PeekMessage loop that processes mouse and keyboard messages. The PatBlt function updates the position of the split bar as it is dragged.

Additional reference words: 3.00 3.10 softlib SPLITWIN.ZIP synchronize
KBCategory:
KBSubcategory: UsrWinMove

INF: Cannot Destroy Default Windows Help Menus and Buttons

Article ID: Q86264

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

A help file built for version 3.1 of the Microsoft Windows Help application can extend the Windows Help application via a number of built-in macro commands. These macros allow the help file author to create a new menu item or button and to delete an added item once it is no longer needed.

However, one cannot use the DeleteItem and DestroyButton macros to remove a standard menu item from the Windows Help application's menus or to remove a button from the standard button bar.

The standard menu items are fixed by the Windows Help application; however, one can modify the behavior of the standard buttons via the ChangeButtonBinding macro.

For more information on Windows Help macros, see Section 15.4 (starting on page 302) of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 4: Resources" manual or the MACROHLP application provided with the Microsoft Windows Software Development Kit (SDK) in the advanced samples directory (by default, C:\WINDEV\SAMPLES).

Additional reference words: 3.10 WinHelp Compiler HC31 HC31.EXE HCP HCP.EXE
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Algorithm Creates Window Same Size As Full-Screen Window

Article ID: Q36319

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

The following algorithm creates a window the size of the "standard" full-screen window:

```
    int winBorder, cxBorder, cyBorder,
        cxMargin, cxBAMargin, cyBAMargin,
        x, y, cx, cy;

    /* Get the user-defined border width */
    winBorder = GetProfileInt("windows", "BorderWidth", 5);
    /* It must be in the range of 0 < winBorder < 51 */
    if (winBorder < 1) {
        winBorder = 1;
    } else if (winBorder > 50) {
        winBorder = 50;
    }

    /* Get some internal system metrics to determine extra scaling */
    cxBorder = GetSystemMetrics (SM_CXBORDER);
    cyBorder = GetSystemMetrics (SM_CYBORDER);
    cxMargin = (cxBorder * winBorder) + cxBorder;

    /* Byte align the border */
    cxBAMargin = (((cxMargin + 7) & 0xFFF8) - cxMargin);
    cyBAMargin = cxBAMargin * cyBorder / cxBorder;

    x  = cxBAMargin;
    y  = cyBAMargin;
    cx = CW_USEDEFAULT;
    cy = 0;

    hWnd = CreateWindow(szAppName, szTitle, WS_OVERLAPPEDWINDOW,
                        x, y, cx, cy,
                        NULL, NULL, hInstance, NULL);
```

Additional reference words: 2.03 2.10 2.x 3.00 3.10 3.x
KBCategory:
KBSubcategory: UsrWinGeneral

INF: Guidelines for Allocating Instance (Per-Window) Data
Article ID: Q76103

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

If an application creates multiple windows of a single window class, that application must carefully manage data for those windows. Undesired results may occur when windows share the same global data because modifications of that data will affect all windows.

If an application has a window that normally relies on global data and the application creates multiple instances of that window, whatever was kept in global data must be kept in instanced data. The optimal method is to store a private data structure in global or local memory and to store a handle to that memory in the window's so-called "extra" bytes. Alternatively, the window may store the handle in its property list.

This article discusses the concerns and methods for storing extra data with windows using both window extra bytes and property lists.

More Information:

1. Application and Window Instances, Sharing Data

The word "instance" usually refers to a particular copy of an application that is currently running in the system. The term "instance" means "copy of data"; each instance of an application has a separate data segment containing global and static data, the application's stack, and the application's local heap. Different instances of the same application have only very limited access to each other's data (through the use of the GetInstanceData function).

Every window is an "instance" of a particular window class. A window is represented by a data structure allocated from USER's local heap (identified by hWnd) that contains information specific to that window, such as its window style, and a pointer to its window procedure. Since multiple windows of the same class may reside in the same application instance and share the same window procedure, those windows freely share all global data in the application's data segment. Therefore, a particular window should never depend on a global variable.

Consider a case in a text editing program based on the Windows multiple document interface (MDI) where each MDI child window maintains information about the file being edited in that window. If

each window uses the same global variables to store page numbers, file names, and text length, only one window would display correct information. As soon as the user loads another file or performs an operation that changes any global variable, all windows are affected. The effects may not be apparent until a repaint occurs on the other windows, making the problem extremely difficult to track down. To avoid this situation, keep any window-specific data separately attached to each window class, isolating that data. Such data may be stored either in window extra bytes or property lists. The most efficient and fastest method is to allocate extra memory for data, and store that memory handle in window extra bytes. Property lists are most useful when dealing with preregistered window classes or with temporary data.

2. Window Extra Bytes and Extra Allocations

An application specifies window extra bytes for a class in the cbWndExtra field of the WNDCLASS structure prior to calling the RegisterClass function. Windows allocates these bytes from USER's local heap in addition to the window's data structure. An application reads these extra bytes with the GetWindowWord and GetWindowLong functions and modifies them with the SetWindowWord and SetWindowLong functions.

Extra bytes are a very scarce resource because they reside in USER's local heap, and therefore reduce available system resources. Always minimize an application's use of window extra bytes as much as possible.

There is a myth that the GetWindowWord and GetWindowLong calls are extremely fast, which serves as an excuse to use a large number of window extra bytes. While these functions are highly optimized, they still need to perform necessary checks on the window handle and trap special cases. They do more than a simple lookup in USER's heap, and both require a far call. Only when an application requires a single WORD or DWORD should it store that data directly in extra bytes.

The most efficient method for allocating more than 4 extra bytes is to declare a single HANDLE in the window's extra bytes, allocate an additional data structure when creating the window, and store that handle in the window extra bytes. This method minimizes the use of extra bytes and provides very fast access to the entire data structure. If the data is allocated as local FIXED memory, there is no need to even lock the data before using it.

The best time to allocate this memory is while processing the WM_NCCREATE message; a window receives this message only once, very soon after creation. The code to perform this allocation might resemble the following:

```
   case WM_NCCREATE:
       pExtraData=(typecast)LocalAlloc(LPTR, cbData);
       SetWindowWord(hWnd, 0, (WORD)pExtraData);
       break;
```

Here, cbData is the number of bytes in the extra data structure, and

the LPTR parameter to the LocalAlloc function specifies the LMEM_FIXED and LMEM_ZEROINIT flags. Use the number 0 as the index for the SetWindowWord call because 0 references the start of the extra bytes. Note also that (typecast) is a placeholder for the appropriate pointer type of the data structure.

The best time to free this memory is while processing the WM_NCDESTROY message. The code to perform this step might resemble the following:

```
    case WM_NCDESTROY:
        pExtraData=GetWindowWord(hWnd, 0);

        if ((typecast)NULL!=pExtraData)
            {
            //Clean out the old handle.
            SetWindowWord(hWnd, 0, 0);
            LocalFree((HANDLE)pExtraData);
            }
        break;
```

2.1 Using Extra Data in the Window Procedure

To access the extra data stored in separately allocated memory, a window must retrieve a pointer to the data before processing any message that requires that data.

If the application allocated memory as LMEM_FIXED, retrieving a pointer to the extra data structure requires only a single call to GetWindowWord, as follows:

```
   pExtraData=(typecast)GetWindowWord(hWnd, 0);
```

From this point on, accessing ANY field in the data structure is extremely fast because it requires only a dereference operation on the pointer. The following is a sample reference of the extra data:

```
   if (0==pExtraData->cMouseMoves)
       {
       ...
       }
```

The main advantage of this method over extended use of window extra bytes is that it uses only a single call to GetWindowWord and SetWindowWord functions to read and write ANY data in the entire data structure. Many calls to the GetWindowWord and SetWindowWord functions are necessary if this structure was stored entirely in extra bytes (unless the structure itself is only 2 bytes long).

In addition, maintaining the data structure is much easier when it is defined as a C structure and not as a large group of offsets for the GetWindowWord and SetWindowWord functions. Maintaining offsets requires carefully defining constants based on the sizes of the data types involved, and defining reasonably symbolic names to keep the source code readable. Defining a C structure is much easier and lets the C compiler determine the size of each field.

2.2 Local Versus Global Memory

Local memory from the application's heap is best suited for the purposes of instanced data, unless heap space is already a scarce commodity. Local allocations are very inexpensive as far as system memory is concerned. In addition, handles to local memory allocated with LMEM_FIXED are directly usable as near pointers.

In Windows version 3.x, each global allocation (through the GlobalAlloc function) uses a selector from the local descriptor table (LDT). Only 8192 are selectors available in Windows version 3.x enhanced mode and only 4096 are available in Windows 3.x standard mode. If an application creates hundreds of instances of a window that allocates a global segment, it will unnecessarily deplete the number of free selectors available to all applications in the system. For more information on using selectors, please query on the following words in the Microsoft Knowledge Base:

   prod(winsdk) and protect mode and handle limits

An application can conserve global handles by allocating a large block of memory and performing local allocations within the block. This technique, known as multiple FAR heaps, is described in Chapter 18 of the book titled "Windows 3.0 Power Programming Techniques" by Paul Yao and Peter Norton (Bantam Computer Books). Another good reference is the article "Improve Windows Application Memory Use with Subsegment Allocation and Custom Resources" in the January 1991 issue of the "Microsoft Systems Journal" (volume 6, number 1).

3. Properties

Properties allow an application to attach extra data to ANY window and do not require the application to process any messages for that window or modify the window's data in any way. An application attaches, retrieves, and deletes properties through the SetProp, GetProp, and RemoveProp functions, respectively.

Properties are most useful when an application needs to attach data to any of Windows's predefined classes, such as dialog boxes and controls, because such windows have no extra bytes available for application use. Properties are also very useful for temporary data that needs to be attached to a window for only a portion of that window's lifetime.

Using properties is relatively slow and has the added penalty that each property is limited to a WORD (which is very inconvenient for large structures). They also use space in the USER's local heap, and therefore consume valuable system resources. Follow the same guidelines for properties as for windows extra bytes, that is, allocate extra memory and store the handle as a single property.

If the application may use extensive properties for dialog boxes or standard Windows controls, consider one of the alternatives described below.

3.1 Alternatives to Property Lists for Dialog Boxes

If an application needs to attach data to a dialog box, consider using a private dialog class. The application registers this class and provides a window procedure similar to the way it would for any other type of application-specific window. A private dialog class requires at least DLGWINDOWEXTRA window extra bytes; the application must add DLGWINDOWEXTRA to any additional space for a memory handle. The extra memory can be allocated and freed using the same messages described in Section 2 above. Note that the offset of the memory handle for the GetWindowWord and SetWindowWord functions is DLGWINDOWEXTRA instead of 0 (zero).

For more information on private dialog classes, please query on the following words in the Microsoft Knowledge Base:

    prod(winsdk) and defdlgproc

3.2 Alternatives to Property Lists for Predefined Control Windows

Most of Windows's predefined controls already use some number of window extra bytes however they do not reserve any space for application-specific use. An application can, however, create a superclass and specify space for a HANDLE to extra data.

Superclassing is the process of retrieving a window's class information, modifying the appropriate fields, and registering the new class under a different name. The following six steps are involved in this process:

1. Call the GetClassInfo function using the hWnd of an existing window of the class to superclass. This fills a WNDCLASS structure with information for that class.

2. Change the hInstance field of the WNDCLASS structure to the hInstance of the application.

3. Save the old lpfnWndProc value and store the address of the superclass procedure in that field. Class extra bytes are a suitable place to save the old value.

   Pass all messages not processed by the superclass procedure to this original lpfnWndProc and be sure to export the superclass procedure.

4. Save the value of cbWndExtra (again, class extra bytes are suitable for this purpose) and add sizeof(HANDLE) to that field. Do NOT simply store sizeof(HANDLE) or the control will break. The superclass MUST use the existing value of cbWndExtra as the offset to GetWindowWord and SetWindowWord functions when manipulating the extra memory handle.

5. Change the class name to something unique. Windows will not reregister a class if one already exists by that name.

6. Call the RegisterClass function with the new WNDCLASS structure.

Allocate extra memory for windows of this superclass using the same method described above in Section 2.

Additional reference words: 3.00 3.10 1.x 2.x 3.x instance control
KBCategory:
KBSubcategory: UsrWinGetwword

INF: Using the DeferWindowPos Family of Functions

Article ID: Q87345

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1
  and 3.0

Summary:

In the Microsoft Windows graphical environment, an application can use the BeginDeferWindowPos, DeferWindowPos, and EndDeferWindowPos functions when it moves or sizes a set of windows simultaneously. Using these functions avoids unnecessary screen painting, which would occur if the windows were moved individually.

The eighth parameter to the DeferWindowPos function can be any one of eight flag values that affect the size and position of each moved or sized window. One of the flags, SWP_NOREDRAW, disables repainting and prevents Windows from displaying any changes to the screen. This flag effects both the client and nonclient areas of the window. Any portion of its parent window uncovered by the move or size operation must be explicitly invalidated and redrawn.

If the moved or sized windows are child windows or pop-up windows, then the SWP_NOREDRAW flag has the expected effect. However, if the window is an edit control, a combo box control, or a list box control, then specifying SWP_NOREDRAW has no effect; the control is drawn at its new location and its previous location is not erased. This behavior is caused by the manner in which these three control classes are painted. Buttons and static controls function normally.

To work around this limitation and move a group of edit, list box, and combo box controls in a visually pleasing manner, perform the following three steps:

1. Use the ShowWindow function to hide all of the controls.

2. Move or size the controls as required with the MoveWindow and
   SetWindowPos functions.

3. Use the ShowWindow function to display all of the controls.

Additional reference words: 3.00 3.10 combobox listbox
KBCategory:
KBSubcategory: UsrWinMove

INF: Minimizing the MS-DOS Executive Window

Article ID: Q61555

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

To minimize the MS-DOS Executive window, you first need to get the handle to this window. To get the handle to the MS-DOS Executive window, you need to use the EnumWindows() function call and search for the string "MS-DOS Executive". Once you obtain this handle, you can make a ShowWindow() call with the handle to the MS-DOS Executive window and either the SW_MINIMIZE, SW_SHOWMINIMIZED, or SW_SHOWMINNOACTIVE flag. For information on these flags, please refer to Page 479 of the "Microsoft Windows Software Development Kit Programmer's Reference" manual.

Additional reference words: 3.00
KBCategory:
KBSubcategory: UsrWinMinmax

INF: Both Windows in SetParent() Call Must Belong to Same Task

Article ID: Q89563

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1
  and 3.0

Summary:

An application can use the SetParent() function to change the parent window for a pop-up, overlapped, or child window. However, the windows identified by the hwndChild and hwndNewParent parameters must both be created by the same instance of the same application.

Additional reference words: 1.x 2.x 3.00 3.10
KBCategory:
KBSubcategory: UsrWinGeneral

PRB: GetWindowPlacement Function Always Returns an Error

Article ID: Q89569

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

SYMPTOMS
   When an application developed for the Microsoft Windows graphical
   environment uses the GetWindowPlacement() function to retrieve the
   show state and position information for a window, the function
   always returns FALSE, indicating an error.

CAUSE
   The length member of the specified WINDOWPLACEMENT data structure
   is not initialized.

RESOLUTION
   Initialize the length member and call the GetWindowPlacement()
   function as follows:

```
      BOOL             bResult;
      WINDOWPLACEMENT  lpWndPl;

      lpWndPl.length = sizeof(WINDOWPLACEMENT);
      bResult = GetWindowPlacement(hWnd, &lpWndPl);
```

More Information:

The need to initialize the length member of the WINDOWPLACEMENT
structure is documented on page 422 of the Microsoft Windows Software
Development Kit (SDK) "Programmer's Reference, Volume 3: Messages,
Structures, and Macros" manual for version 3.1. This information is
not listed in the documentation for the GetWindowPlacement() function on
page 479 of the "Programmer's Reference, Volume 2: Functions" manual.

Additional reference words: 3.10 docerr
KBCategory:
KBSubcategory: UsrWinGeneral

INF: Changing the Parent of a Child Window Using SetParent

Article ID: Q75001

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

PARENT is a sample application in the Software/Data Library that demonstrates how to dynamically change the parent of a child window by using the SetParent function.

Note that this technique is very risky when applied to pop-up windows. Pop-up windows do not have a parent, instead they have an owner, which is a different relationship. Trying to change the parent of a pop-up window can cause the application or Windows to crash.

PARENT can be found in the Software/Data Library by searching on the keyword PARENT, the Q number of this article, or S13130. PARENT was archived using the PKware file-compression utility.

Additional reference words: 3.00 3.10 3.x softlib
KBCategory:
KBSubcategory: UsrWinRareMisc

INF: Transparent Windows

Article ID: Q92526

The information in this article applies to:

 - Microsoft Windows Software Development Kit for Windows versions 3.1
   and 3.0

Summary:

Microsoft Windows version 3.1 does not support fully functional transparent windows.

More Information:

If a window is created using CreateWindowEx() with the WS_EX_TRANSPARENT style, windows below it at the position where the original window was initially placed are not obscured and show through. Moving the WS_EX_TRANSPARENT window, however, results in the old window background moving to the new position, because Windows does not support fully functional transparent windows.

WS_EX_TRANSPARENT was designed to be used in very modal situations and the lifetime of a window with this style must be very short. A good use of this style is for drawing tracking points on the top of another window. For example, a dialog editor would use it to draw tracking points around the control that is being selected or moved.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrWinStyles

INF: Different Ways to Close an Application Under Windows

Article ID: Q77135

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0

Summary:

In the Microsoft Windows graphical environment, an application can close in two different ways. The first method involves the user closing the application, which includes choosing a menu item, double-clicking the application's system menu, or choosing End Task in the Windows Task Manager. The second method involves the user ending the Windows session in the active shell program (either Program Manager or File Manager).

In these two methods, the application receives different sets of messages. If an application must perform certain operations before it can exit properly, it must process the messages associated with each method. This article discusses these messages and how to process them.

More Information:

Method 1

The three sets of actions below are facets of the first method above:

1. When the user double-clicks the application's system menu or chooses Close from the system menu, Windows sends a WM_SYSCOMMAND message to the application with wParam set to SC_CLOSE. If the user enters the ALT+F4 accelerator for Close, Windows posts the WM_SYSCOMMAND message to the application. If the application passes the WM_SYSCOMMAND message to DefWindowProc, the application receives a WM_CLOSE message.

2. If the user chooses End Task in the Windows Task Manager, the Task Manager posts a WM_CLOSE message to the application's main window.

3. If the application has an Exit option on its File menu, it should be coded to generate a WM_CLOSE message.

When an application receives a WM_CLOSE message, it must determine whether the user has modified any files without saving the changes. If so, the application must inform the user that changes have not been saved and provide an opportunity to save the files, to close the application without saving the files, or to cancel the application termination request.

If the application is to close, it must call the DestroyWindow function to destroy the main window. If the user is not ready to close the application, the application must not pass the WM_CLOSE to

DefWindowProc because it will call DestroyWindow.

DestroyWindow recursively destroys all child windows of the main window and all windows owned by the main window. If the application has created windows that are neither children of nor owned by the main window, the application must explicitly destroy these windows.

Method 2

In the second method listed above, when the user ends the Windows session from the active shell program (Program Manager or File Manager), Windows sends a WM_QUERYENDSESSION message to all tasks in the system. An application can process this message to determine when the user terminates the Windows session. When an application receives a WM_QUERYENDSESSION message, if files have been changed and the changes have not been saved, the application must inform the user and confirm the request to close the application. If the application receives confirmation, it must return TRUE after processing the WM_QUERYENDSESSION message. If the user does not confirm closing the application, it must return FALSE to cancel closing Windows.

If any application cancels closing Windows in response to a WM_QUERYENDSESSION message, Windows sends a WM_ENDSESSION message with wParam set to FALSE for all applications. This informs each application that the Windows session will continue.

If all applications return TRUE after processing the WM_QUERYENDSESSION message, Windows sends each application a WM_ENDSESSION message with wParam set to TRUE. This informs the application that the current Windows session will end very shortly. After the WM_ENDSESSION message, an application receives neither a WM_CLOSE nor a WM_DESTROY message.

In summary, if an application must perform operations before it can exit, such as saving files, the application must be prepared for both the application close and Windows shutdown cases.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: UsrWinDestwin

INF: Adding to or Removing Windows from the Task List

Article ID: Q99800

Summary:

There are no functions included in the Windows Software Development Kit (SDK) to add or remove windows from the task list. All top-level windows (that is, windows without parents) that are visible will automatically appear in the task list. A window can be removed from the task list by making it hidden. Call ShowWindow() with the SW_HIDE parameter to hide the window. To make it visible again, call ShowWindow() with the appropriate parameter such as SW_SHOW or SW_SHOWNORMAL.

Additional reference words: 3.00 3.10 tasklist
KBCategory:
KBSubcategory: UsrWinShowhide

INF: Safe Subclassing
Article ID: Q101180

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1

SUMMARY
=======

This article describes subclassing, how it is done, and the rules that should be followed to make subclassing safe. Both instance and global subclassing are covered. Superclassing is described as an alternative to global subclassing.

MORE INFORMATION
================

Subclassing Defined

Subclassing is a technique that allows an application to intercept messages destined for another window. An application can augment, monitor, or modify the default behavior of a window by intercepting messages meant for another window. Subclassing is an effective way to change or extend the behavior of a window without redeveloping the window. Subclassing the default control window classes (button, edit, list box, combo box, static, and scroll bar controls) is a convenient way to obtain the functionality of the control and to modify its behavior. For example, if a multiline edit control is included in a dialog box and the user presses the ENTER key, the dialog box closes. By subclassing the edit control, an application can have the edit control insert a carriage return and linefeed into the text without exiting the dialog box. An edit control does not have to be developed specifically for the needs of the application.

The Basics

The first step in creating a window is registering a window class by filling a WNDCLASS structure and calling RegisterClass. One element of the WNDCLASS structure is the address of the window procedure for this window class. When a window is created, the Microsoft Windows graphical environment takes the address of the window procedure in the WNDCLASS structure and copies it to the new window's information structure. When a message is sent to the window, Windows calls the window procedure through the address in the window's information structure. To subclass a window, you substitute a new window procedure that receives all the messages meant for the original window by substituting the window procedure address with the new window procedure address.

When an application subclasses a window, it can take three actions

with the message: (1) pass the message to the original window procedure; (2) modify the message and pass it to the original window procedure; (3) not pass the message. The application subclassing a window can decide when to react to the messages it receives. The application can process the message before, after, or both before and after passing the message to the original window procedure.

Types of Subclassing

The two types of subclassing are instance subclassing and global subclassing. Instance subclassing is subclassing an individual window's information structure. With instance subclassing, only the messages of a particular window instance are sent to the new window procedure. Global subclassing is replacing the address of the window procedure in the WNDCLASS structure of a window class. All subsequent windows created with this class have the substituted window procedure's address. Global subclassing affects only windows created after the subclass has occurred. If any windows exist of the window class that is being globally subclassed at the time of the subclass, the existing windows are not affected by the global subclass. If the application needs to affect the behavior of the existing windows, the application must subclass each existing instance of the window class.

Instance Subclassing

The SetWindowLong function is used to subclass an instance of a window. The application must have the procedure-instance address of the subclass function. The subclass function is the function that receives the messages from Windows and passes the messages to the original window procedure. To get the procedure-instance address of the subclass function, the application calls MakeProcInstance. If the subclass function resides in a dynamic-link library (DLL), the application does not need to call MakeProcInstance to get the procedure-instance address. The subclass function must be exported in the application's or the DLL's module definition file.

The application subclassing the window calls SetWindowLong with the handle to the window the application wants to subclass, the GWL_WNDPROC option (defined in WINDOWS.H), and the procedure-instance address of the new subclass function. SetWindowLong returns a DWORD, which is the address of the original window procedure for the window. The application must save this address to pass the intercepted messages to the original window procedure and to remove the subclass from the window. The application passes the messages to the original window procedure by calling CallWindowProc with the address of the original window procedure and the hWnd, Message, wParam, and lParam parameters used in Windows messaging. Usually, the application simply passes the arguments it receives from Windows to CallWindowProc.

The application also needs the original window procedure address for removing the subclass from the window. The application removes the subclass from the window by calling SetWindowLong again. The application passes the address of the original window procedure with the GWL_WNDPROC option and the handle to the window being subclassed. The following code subclasses and removes a subclass to an edit

control:

```
LONG FAR PASCAL SubClassFunc(HWND hWnd,WORD Message,WORD wParam,
      LONG lParam);
FARPROC lpfnOldWndProc;
HWND hEditWnd;

//
// Create an edit control and subclass it.
// The details of this particular edit control are not important.
//
hEditWnd = CreateWindow("EDIT", "EDIT Test",
                        WS_CHILD | WS_VISIBLE | WS_BORDER ,
                        0, 0, 50, 50,
                        hWndMain,
                        NULL,
                        hInst,

                        NULL);
//
// Now subclass the window that was just created.
//
lpfnSubClassProc=MakeProcInstance((FARPROC) SubClassFunc,hInst);
lpfnOldWndProc =
   (FARPROC)SetWindowLong(hEditWnd, GWL_WNDPROC, (DWORD)
      lpfnSubClassProc);
.
.
.
//
// Remove the subclass for the edit control.
//
SetWindowLong(hEditWnd, GWL_WNDPROC, (DWORD) lpfnOldWndProc);

//
// Here is a sample subclass function.
//
LONG FAR PASCAL SubClassFunc(   HWND hWnd,

               WORD Message,
               WORD wParam,
               LONG lParam)
{
   //
   // When the focus is in an edit control inside a dialog box, the
   //  default ENTER key action will not occur.
   //

   if ( Message == WM_GETDLGCODE )
       return DLGC_WANTALLKEYS;

   return CallWindowProc(lpfnOldClassProc, hWnd, Message, wParam,
                         lParam);
}
```

Potential Pitfalls

Instance subclassing is generally safe, but observing the following rules ensures safety. When subclassing a window, you must know what owns the window's behavior. For example, Windows owns all controls it supplies; applications own all windows they define. Subclassing can be done on any window in the system; however, when an application subclasses a window that the application does not own, the application must ensure that the subclass function does not break the original behavior of the window. Because the application does not control the window, it should not rely on any information about the window that the owner might change in the future. A subclass function should not use the extra window bytes or the class bytes for the window unless it knows exactly what they mean and how the original window procedure uses them. Even if the application knows everything about the extra window bytes or the class bytes, it should not use them unless the application owns the window. If an application uses the extra window bytes of a window that another application owns and the owner decides to update the window and change some aspect of the extra bytes, the subclass procedure is likely to fail. For this reason, Microsoft suggests that you not subclass the control classes. Windows owns the controls it supplies, and aspects of the controls might change from one version of Windows to the next. If your application must subclass a control supplied by Windows, it may need to be updated when a new version of Windows is released.

Because instance subclassing occurs after a window is created, the application subclassing the window cannot add any extra bytes to the window. Applications that subclass windows should store any data needed for an instance of the subclassed window in the window's property list. The SetProp function attaches properties to a window. The application calls SetProp with the handle to a window, a string identifying the property, and a handle to the data. The handle to the data is usually obtained with a call to either LocalAlloc or GlobalAlloc. When the application uses the data in a window's property list, the application calls the GetProp function with the handle to the window and the string that identifies the property. GetProp returns the handle to the data that was set with SetProp. When the application is finished with the data or when the window is to be destroyed, the application must remove the property from the property list by calling the RemoveProp function with the handle to the window and the string identifying the property. RemoveProp returns the handle to the data, which the application then uses in a call to either LocalFree or GlobalFree. The Microsoft Windows SDK "Programmer's Reference, Volume 2, Functions" manual describes SetProp, GetProp, and RemoveProp.

When an application subclasses a subclassed window, the subclasses must be removed in reverse of the order in which they were performed.

Global Subclassing

Global subclassing is similar to instance subclassing. The application calls SetClassLong to globally subclass a window class. As it does with instance subclassing, the application needs the procedure-instance address of the subclass function, and the subclass function must be exported in the application's or the DLL's module

definition file. To globally subclass a window class, the application must have a handle to a window of that class. To get a handle to a window in the desired class, most applications create a window of the class to be globally subclassed. When the application removes the subclass, it needs a handle to a window of the type the application wants to subclass, so creating and keeping a window for this purpose is the best technique. If the application creates a window of the type it wants to subclass, the window is usually hidden. After obtaining a handle to a window of the correct type, the application calls SetClassLong with the window handle, the GCL_WNDPROC option (defined in WINDOWS.H), and the procedure-instance address of the new subclass function. SetClassLong returns a DWORD, which is the address of the original window procedure for the class. The original window procedure address is used in global subclassing in the same way it is used in instance subclassing. Messages are passed to the original window procedure in the same way as in instance subclassing, by calling CallWindowProc. The application removes the subclass from the window class by calling SetClassLong again. The application passes the address of the original window procedure with the GCL_WNDPROC option and a handle to the window of the class being subclassed. An application that globally subclasses a control class must remove the subclass when the application finishes.

The following code globally subclasses and removes a subclass to an edit control:

```
LONG FAR PASCAL SubClassFunc(HWND hWnd,WORD Message,WORD wParam,
      LONG lParam);
FARPROC lpfnOldClassWndProc;
HWND hEditWnd;

//
// Create an edit control and subclass it.
// Notice that the edit control is not visible.
// Other details of this particular edit control are not important.
//
hEditWnd = CreateWindow("EDIT", "EDIT Test",
                        WS_CHILD,
                        0, 0, 50, 50,
                        hWndMain,
                        NULL,

                        hInst,
                        NULL);
lpfnSubClassProc=MakeProcInstance((FARPROC) SubClassFunc,hInst);
lpfnOldClassWndProc =
   (FARPROC)SetClassLong(hEditWnd, GCL_WNDPROC, (DWORD)
      lpfnSubClassProc);
.
.
.
//
// To remove the subclass:
//
SetClassLong(hEditWnd, GWL_WNDPROC, (DWORD) lpfnOldClassWndProc);
DestroyWindow(hEditWnd);
```

Potential Pitfalls

Global subclassing has the same limitations as instance subclassing but presents some additional problems. The application should not attempt to use the extra bytes for either the class or the window instance unless it knows exactly how the original window procedure is using them. If data must be associated with a window, the window's properties list should be used in the same way as in instance subclassing. Global subclassing is dangerous when used with the Windows-supplied control classes, especially if more than one application globally subclasses a control class.

The following sequence of events illustrates this problem. Application A globally subclasses the edit control class and stores the original window procedure's address. Application B then also globally subclasses the edit control class. When application B calls SetClassLong with the address of application B's subclass function, SetClassLong returns the address of application A's subclass function instead of the address of the original window procedure for the window class. If application A removes the subclass from the edit control class, it replaces the original window procedure's address in the WNDCLASS structure. From this point on, application B's subclass is effectively removed. New edit controls have the original window procedure's address as their window procedure. If application B then removes its subclass of the edit control class, even more damaging events occur. Application B replaces the original window procedure's address in the WNDCLASS structure with the address of application A's subclass function. New edit controls now attempt to use application A's subclass function. If application A is still loaded, its subclass of the edit control class has effectively been reinstalled, even though application A is not aware of this. If application A is no longer loaded, Windows version 3.0 still attempts to call a procedure at the address of application A's subclass function; this may cause a FatalExit, a UAE, or the system to hang. Windows version 3.1 does not allow this to happen.

Because of the dangers of globally subclassing the control classes, global subclassing should be done only on application-specific window classes. If your application could benefit from globally subclassing a control class, your application should use an alternative technique called superclassing.

SUPERCLASSING

Subclassing a window class causes messages meant for the window procedure to be sent to the class function. The class function then passes the message to the original window procedure. Superclassing creates a new window class. The new window class uses the window procedure from an existing class to give the new class the functionality of the existing class. The superclass is based on some other window class, known as the base class. Frequently the base class is a Windows-supplied control class, but it can be any window class. Do not superclass the scroll bar control class because Windows uses the class name to produce the correct behavior for scroll bars.

The superclass has its own window procedure, the superclass procedure, which can take the same actions a subclass procedure can. The superclass procedure can take three actions with the message: (1) pass the message directly to the original window procedure; (2) modify the message before passing it to the original window procedure; (3) not pass the message. The superclass can react to the message before, after, or both before and after passing the message to the original window procedure.

Unlike a subclass procedure, a superclass procedure receives create (WM_NCCREATE, WM_CREATE, and so on) messages from Windows. The superclass procedure can process these messages, but it must also pass these messages to the original base-class window procedure so that the base-class window procedure can initialize. The application calls GetClassInfo to base a superclass on a base class. GetClassInfo fills a WNDCLASS structure with the values from the base class's WNDCLASS structure. The application that is superclassing the base class then sets the hInstance field of the WNDCLASS structure to the instance handle of the application. The application must also set the WNDCLASS structure's lpszClassName field to the name it wants to give this superclass. If the base class has a menu, the application superclassing the base class must supply a new menu that has the same menu IDs as the base class's menu. If the superclass intends to process the WM_COMMAND message and not pass the message to the base class's window procedure, the menu does not have to have corresponding IDs. GetClassInfo does not return the lpszMenuName, lpszClassName, or hInstance field of the WNDCLASS structure.

The last field that must be set in the superclass's WNDCLASS structure is the lpfnWndProc field. GetClassInfo fills this field with the original class window procedure. The application must save this address so that it can pass messages to the original window procedure with a call to CallWindowProc. The application must put the address of its subclass function into the WNDCLASS structure. This address is not a procedure-instance address because RegisterClass gets the procedure-instance address. The application can modify any other fields in the WNDCLASS structure to suit the application's needs.

The application can add to both the extra class bytes and the extra window bytes because it is registering a new class. The application must follow two rules when doing this: (1) The original extra bytes for both the class and the window must not be touched by the superclass for the same reasons that an instance subclass or a global subclass should not touch these extra bytes; (2) if the application adds extra bytes to either the class or the window instance for the application's own use, it must always reference these extra bytes relative to the number of extra bytes used by the original base class. Because the number of bytes used by the base class may be different from one version of the base class to the next, the starting offset for the superclass's own extra bytes is also different from one version of the base class to the next.

After the WNDCLASS structure is filled, the application calls RegisterClass to register the new window class. Windows of this class can now be created and used.

Additional reference words: 3.1 3.10

KBCategory:
KBSubcategory: UsrWinSubclass UsrCtlSubclass