Programmer's Reference

Microsoft® MODULAR WINDOWS™

SOFTWARE DEVELOPMENT KIT

Programmer's Reference

Microsoft® Modular Windows™ Software Development Kit

Microsoft Corporation

Document No. MM37744-1192
Printed in the United States of America.

Contents

Introduction

This manual, *Microsoft Modular Windows Programmer's Reference*, describes the application programming interface (API) supported by the Microsoft® Modular Windows™ Operating System. It also shows how to use the programming tools included in the Modular Windows Software Development Kit (SDK).

How to Use This Manual

This manual provides both an overview of the Microsoft Modular Windows API and an alphabetical directory of functions, messages, and data structures that are new for Modular Windows. It also includes a directory of the Microsoft Windows 3.1 functions that have changed or are not supported in Modular Windows.

You should read the overview chapters first, and then use the directory chapters as a reference when programming applications for Modular Windows. The following is a summary of the chapters in this manual:

- Chapter 1, "User-Interface Controls," gives an overview of the differences between the Windows 3.1 user interface and the Modular Windows user interface and describes the new API for user-interface controls.
- Chapter 2, "Hand-Control Services," gives an overview of using a hand control as an input device and describes the new API for the hand control.
- Chapter 3, "Video Services," describes the Modular Windows display drivers and gives an overview of the new API for displaying images and directly accessing video memory.
- Chapter 4, "Core API and Extension Libraries Support," is a list of the Windows 3.1 functions that have changed or are not supported in Modular Windows.
- Chapter 5, "Function Directory," is an alphabetical reference to new Modular Windows functions and macros.
- Chapter 6, "Message Directory," is an alphabetical reference to new Modular Windows messages.
- Chapter 7, "Data Structures," is an alphabetical reference to new Modular Windows data structures.

- Chapter 8, "File Formats," provides details on new image file formats for Modular Windows.
- Chapter 9, "Tools," is a guide to using the programming tools included in the Modular Windows SDK.
- Appendix A, "VIS Memory-Cartridge Services," describes the API for the removable Save-It™ memory cartridge for the Tandy® Video Information System™ (VIS™) players.
- Appendix B, "VIS Programming Notes," provides details about programming for VIS players.

Document Conventions

The following conventions are used throughout this reference to define syntax:

Convention	Meaning
Bold text	Denotes a term or character to be typed literally, such as a function name (**CreateWindow**), data-structure name (**WNDCLASS**), or resource-definition statement (**ICON**). You must type these terms exactly as shown.
Italic text	Denotes a placeholder or variable; you must provide the actual value. For example, the statement **SetCursorPos**(*X*,*Y*) requires you to substitute values for the *X* and *Y* parameters.
[ ]	Encloses optional parameters.
|	Separates an either/or choice.
. . .	Specifies that the preceding item can be repeated.
. . .	Represents an omitted portion of sample code.

The following text conventions are also used in this reference:

Convention	Meaning
SMALL CAPITALS	Indicates the names of keys, key sequences, and key combinations—for example, ALT+SPACEBAR.
ALL CAPITALS	Indicates filenames and paths, most type and structure names (which are also bold), and constants.
`monospace`	Indicates C- and assembly-language source-code statements.

CHAPTER 1

User-Interface Controls

The standard user-interface model of drop-down menus and multiple overlapping windows was designed to be used with a mouse and a keyboard as input devices. Much of this standard user-interface support is not available in Modular Windows—it has been replaced with a simpler user interface optimized for use with a hand control and designed to be displayed on a television.

This chapter discusses the Microsoft Modular Windows user interface and how it differs from the standard menu-based user interface, introduces the TV user-interface controls, and shows how you can use these controls in an application for TV-based players.

User-Interface Elements Not Available in Modular Windows

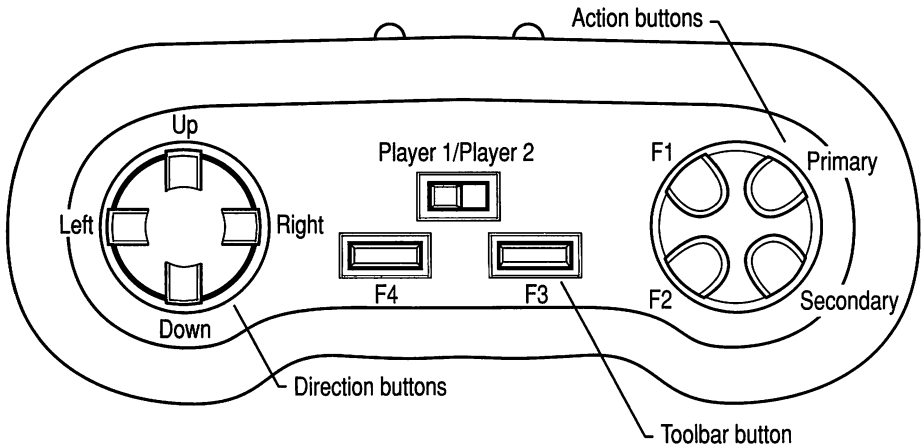
The following list identifies elements of the standard Microsoft Windows 3.1 user interface not available in Modular Windows:

- Menus
- System menu, sizing borders, minimize/maximize buttons, and non-client scroll bars (WS_CAPTION, WS_MAXIMIZEBOX, WS_MINIMIZEBOX, WS_SYSMENU, WS_THICKFRAME, WS_HSCROLL and WS_VSCROLL window styles)
- Controls based on the COMBOBOX and MDICLIENT predefined window control classes (BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR, and STATIC controls are translated to corresponding Modular Windows controls)
- Multiple Document Interface (MDI)
- Common dialog-box library (COMMDLG)

Applications written for TV-based multimedia players must be designed with a different user interface than applications designed for personal computers. Modular Windows provides a rich set of controls and user-interface elements designed to provide both visual and tactile appeal on TV-based players.

Using the Hand Control as an Input Device

The TV user-interface controls are designed for use with a hand control. The following illustration shows the layout of the ten buttons on a hand control:

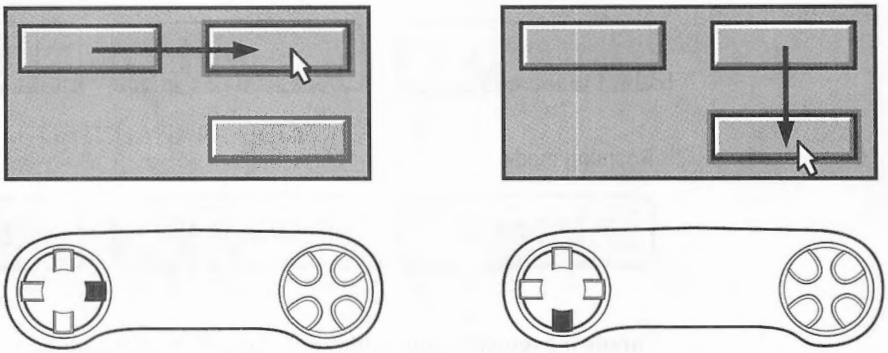


Hand control for TV-based multimedia players

User Input with a Hand Control

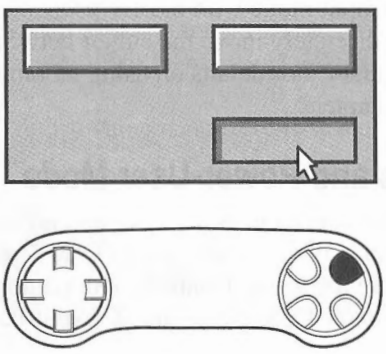
User input with the hand control is a two step process.

1. The user presses the direction buttons to move the cursor between controls, as shown in the following illustration:



Using the direction buttons to move the focus between controls

2. The user then presses the primary action button to initiate an action associated with the control.

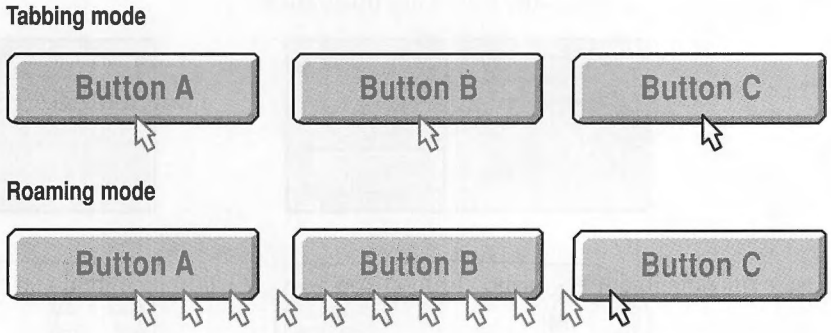


Using the action button to actuate a control

Moving the Focus Using Tabbing and Roaming Modes

Modular Windows provides two modes for moving the cursor between controls: *roaming mode* and *tabbing mode*. In roaming mode, the user positions the cursor over a control by pressing the direction buttons to move the cursor in the corresponding direction. Positioning the cursor in this manner is an iterative process requiring several rounds of horizontal and vertical movement, but it allows the cursor to be accurately positioned in any location on the screen. In tabbing mode, the user discretely moves the focus between controls using the

direction buttons. The operation of tabbing mode is similar to using the TAB key on a keyboard to tab between controls in a standard Windows dialog box. The following illustration shows cursor movement using tabbing mode and roaming mode:



Cursor movement using tabbing mode and roaming mode

Note The default mode is tabbing mode—applications must make a call to the **hcControl** function to enable roaming mode.

Tabbing is the preferred input mode with the TV user-interface controls. It is much easier for users to discretely move the cursor between controls than to iteratively position the cursor. For details on using tabbing mode, see "The Focus Manager," later in this chapter.

Compound Focus and Power-User Mode

Some of the TV user-interface controls have compound focus, or more than one area that can have the focus. For example, each of the scroll arrows in a TV scroll-bar control can have the focus. Controls with compound focus include the TV scroll-bar, TV scroll-pad, TV list-box, and TV spin-button controls.

All of the controls with compound focus have a power-user mode that allows faster input using the hand control. Power-user mode is initiated when the user holds the action button down while pressing one of the direction buttons. As long as the action button is pressed, pressing a direction button not only changes the focus, but also simulates a button press on the action button. For example, with a scroll-bar control, a scroll event occurs each time a direction button is pressed while the action button is down.

Using the Mouse and Keyboard as Input Devices

A mouse and standard PC keyboard are optional input devices for TV-based players. The Modular Windows user-interface controls operate with both a mouse and a keyboard as well as a hand control.

Operation of the user-interface controls with a mouse is identical to the way the controls operate with the hand control in roaming mode—moving the mouse moves the cursor and pressing the left mouse button chooses the control.

If a keyboard is present, it can be used to enter text in edit-box controls. In addition, keyboard keys can be used to simulate hand-control button actions to move the focus and choose controls. The following table lists the keyboard keys that correspond to hand-control buttons:

Keyboard Key	Corresponding Hand-Control Button
UP ARROW key	Up direction button
DOWN ARROW key	Down direction button
LEFT ARROW key	Left direction button
RIGHT ARROW key	Right direction
SPACE BAR	Primary action button

TV User-Interface Controls

Modular Windows provides the following user-interface controls:

Control	Purpose
TV push button	Initiate an action.
TV check box	Select or cancel the selection of an option.
TV radio button	Choose one of several options.
TV group box	Visually associate related controls. TV group boxes don't accept user input.
TV scroll bar	Position information displayed in a window or to select a value from a range of values.
TV gauge	Display position or progress information. TV gauges are display-only controls and don't accept user input.
TV list box	Select an item from a list.

Continued

Control	Purpose
TV scroll pad	Horizontally and vertically position information displayed in a window.
TV spin button	Increment and decrement a value or selection.
TV static control	Display text or icons. TV static controls don't accept user input.
TV show box	Display text and bitmaps within a 3-D frame. TV show boxes don't accept user input.
TV keyboard and TV edit box	Enter text using a hand control.

Applications can use these controls in a dialog box created by using either the **CreateDialog**, **CreateDialogIndirect**, or **DialogBox** function, or as child window controls created with the **CreateWindow** function.

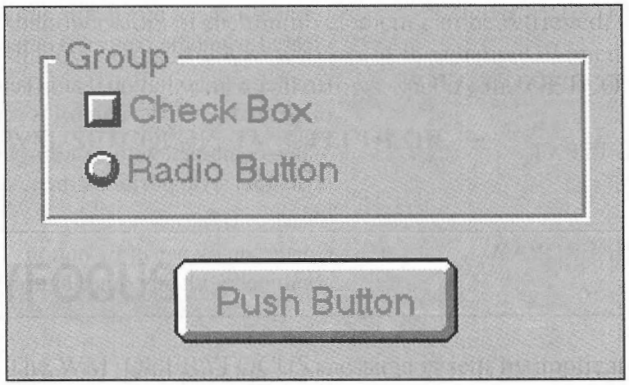
Note Before creating a TV user-interface control or a dialog box that uses a TV user-interface control, you must load the TVUI.DLL library to register the new control classes. To load the library, call a function in the library, such as **tvGetUIFlags**.

TV Button Control (TVBUTTON)

The TV button control is similar to the standard Windows button control with the following changes:

- The appearance is optimized for display on televisions.
- Applications can customize the appearance by changing colors and supplying bitmaps for different button elements.

The following illustration shows the appearance of four styles of TV button controls, the group-box, check-box, radio-button, and push-button styles:



TV button controls: group-box, check-box, radio-button, and push-button

TV Button Styles

The following table lists the control styles for TV button controls. These styles can be used in the *dwStyle* parameter of the **CreateWindow** function when creating a child-window control or in the *style* field of the **CONTROL** resource script statement when creating a dialog template.

Style	Description
BS_PUSHBUTTON	Creates a push button.
BS_STICKYBUTTON	Creates a push button that only changes state when the hand-control action button is pressed. Regular push buttons depress when the action button is pressed and pop back up when the action button is released.
BS_CHECKBOX	Creates a check box. The application must manage selecting and clearing when the control is selected by the user.
BS_RADIOBUTTON	Creates a radio button. The application must manage selecting and clearing when the control is selected by the user.
BS_GROUPBOX	Creates a group box.

Continued

Style	Description
BS_AUTOCHECKBOX	Creates a check box that is automatically selected or cleared when the control is selected by the user.
BS_AUTORADIOBUTTON	Creates a radio button that is automatically highlighted or cleared when the control is selected by the user.
BS_LEFTTEXT	When combined with a radio-button or check-box style, this style positions the text to the left of the button.
BS_OWNERDRAW	Creates an owner-draw button. This style cannot be combined with any other button styles.

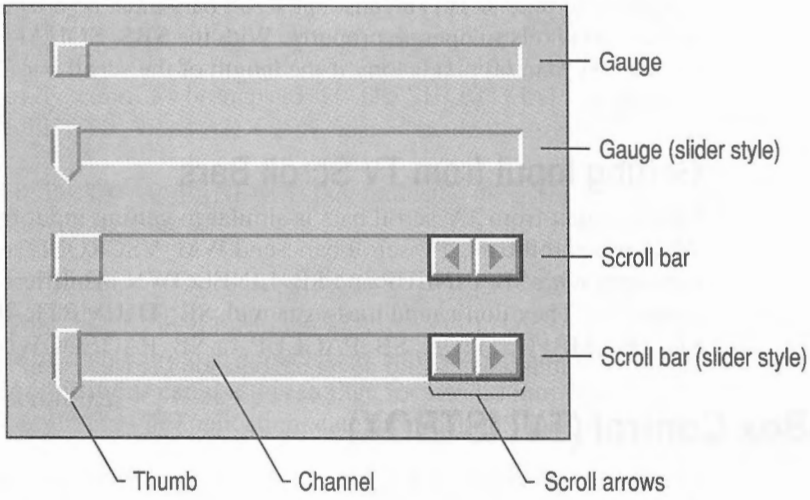
The BS_DEFPUSHBUTTON style is not supported because there is no equivalent to the ENTER key on a hand control. The BS_STICKYBUTTON style is the only new button style for TV button controls.

TV Scroll Bar and TV Gauge Control (TVSCROLLBAR)

The TV scroll-bar control is similar to the standard Windows scroll-bar control with the following changes:

- Non-client scroll bars are not supported. Scroll bars must be created and managed by the application.
- There is a new style of scroll bar called a *gauge*. Gauges don't accept user input.
- Applications can supply bitmaps and change colors to customize the appearance of scroll bars and gauges.

The following illustration shows the appearance of TV scroll-bar controls (only horizontal controls are shown):



TV scroll-bar controls

TV Scroll-Bar Styles

The following table lists the control styles for TV scroll-bar controls:

Style	Description
SBS_HORZ	Creates a horizontal scroll bar.
SBS_VERT	Creates a vertical scroll bar.
SBS_DISPLAYONLY	Creates a gauge. This style must be combined with either the SBS_HORZ or SBS_VERT style.
SBS_SLIDER	When combined with other scroll-bar styles, this style creates a scroll bar or gauge with a thumb that is shaped like a pointer (see the previous illustration).
SBS_SQUARE	Creates a scroll bar with square buttons and square thumb (if SBS_SLIDER specified, the thumb is not affected by SBS_SQUARE).
SBS_THUMBINSIDE	When combined with other scroll-bar styles, creates a scroll bar or gauge with the thumb clipped to the inside of the channel.

TV Scroll-Bar Sizing

Without the SBS_SQUARE style, scroll-bar buttons and the thumb have a fixed length of 20 pixels. This means that scroll bars must be created using a length of at least 60 pixels to operate properly. With the SBS_SQUARE style, scroll bars can be less than 60 pixels long if the length of the scroll bar is at least three times the width.

Getting Input from TV Scroll Bars

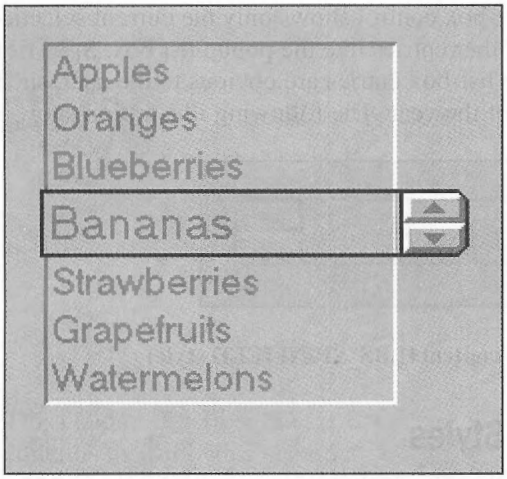
Getting input from TV scroll bars is similar to getting input from standard Windows scroll bars. TV scroll bars send WM_VSCROLL and WM_HSCROLL messages with SB_LINEUP and SB_LINEDOWN identifiers in the *wParam* parameter. They don't send messages with SB_THUMBTRACK, SB_THUMBPOSITION, SB_PAGEUP, or SB_PAGEDOWN identifiers.

TV List-Box Control (TVLISTBOX)

The TV list-box control is similar to the standard Windows list-box control with the following changes:

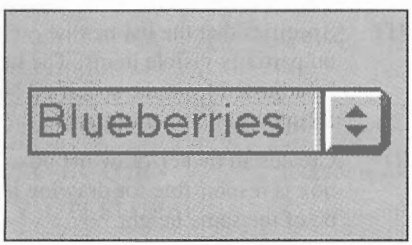
- Instead of keeping list-box entries in a fixed location while moving the highlight, the TV list box keeps the selection area fixed and moves the list-box entries.
- Selection of an entry is indicated by drawing the selected text or bitmap slightly larger than the other entries.
- One item is always selected.
- Multiple-column and multiple-selection styles are not supported.

The following illustration shows the appearance of a standard TV list box (the current selection is "Bananas"):

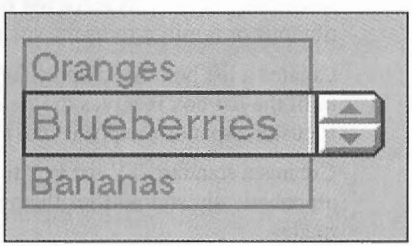


Standard TV list-box control (LBS_STANDARD style)

There are two variations of the standard TV list box: popup and spin field. The popup list-box control displays only the current selection until it gets the focus and is activated with the primary action button. Once activated, it expands to appear like a standard list box. The following illustration shows the appearance of a popup list box before and after it is activated:



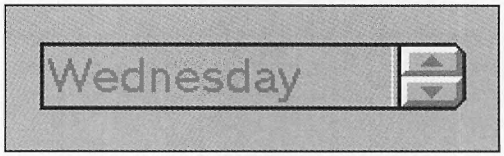
Before activation



After activation

Popup TV list-box control (LBS_POPUP style)

The spin-field list-box control shows only the current selection—it does not expand to show other entries like the popup list box. Spin-field list boxes should be used when the list-box entries are obvious to the user, such as for days of the week or months of the year. The following illustration shows a spin-field list box:



Spin field list-box control (LBS_SPINFIELD style)

TV List-Box Styles

The following table lists the available styles for TV list-box controls:

Style	Description
LBS_HASSTRINGS	Specifies that the list-box items are strings. By default, all list boxes except owner-draw list boxes have this style.
LBS_NOTIFY	Specifies that the list box notify the owner when a selection is made and when the list box gets or loses the input focus.
LBS_NOINTEGRALHEIGHT	Specifies that the list box be created such that it contains no partially visible items. The height of the list box will be adjusted from the specified height to accommodate an integral number of items.
LBS_OWNERDRAWFIXED	Creates an owner-draw list box. The owner of the list box is responsible for drawing its items, which must all be of the same height.
LBS_SORT	Specifies that the list box sort items alphabetically.
LBS_SPINFIELD	Creates a list box that displays only the current selection. Use this style for list boxes containing items obvious or familiar to users, such as months of the year.
LBS_POPUP	Creates a list box that displays only the current selection until the list box receives the focus. Once it receives the focus, it expands to appear like a standard list box.
LBS_STANDARD	Creates a standard TV list box that sorts strings alphabetically and notifies the owner when a selection is made.

TV List-Box Messages

There are two types of messages associated with list-box controls:

- Standard list-box messages are sent by applications to a list-box control to initiate or request an action, such as adding a string to the list box or getting the text of a given item in the list box.
- Notification codes are sent to applications to notify the application of a user action, such as selecting an item in the list box. Notification codes are received in the high-order word of the *lParam* parameter of a WM_COMMAND message.

Standard List-Box Messages

The following table lists the messages that applications can send to TV list boxes:

Message	Description
LB_ADDSTRING	Adds a string to a list box.
LB_DELETESTRING	Deletes a string from a list box.
LB_FINDSTRING	Searches a list box for a given string.
LB_GETCOUNT	Gets the number of items in a list box.
LB_GETCURSEL	Gets the index of the currently selected item in a list box.
LB_GETITEMDATA	Gets a value associated with a list-box item.
LB_GETITEMRECT	Gets the bounding rectangle for a list-box item.
LB_GETPOPUPRECT	Retrieves the bounding rectangle of a popup list box as it appears when activated.
LB_GETSELAREA	Retrieves the current selection area.
LB_GETTEXT	Gets the text of a list-box item.
LB_GETTEXTLEN	Gets the length of the text in a list-box item.
LB_INSERTSTRING	Inserts a string into a list box.
LB_RESETCONTENT	Removes all items from a list box.
LB_SETCURSEL	Changes the current selection in a list box.
LB_SETITEMDATA	Associates a value with a list-box item.
LB_SETITEMHEIGHT	Changes the height of items in an owner-draw list box.
LB_SETSELAREA	Sets the current selection area.
LB_SETTABSTOPS	Sets tab stops for all items in a list box.

Standard Windows list-box messages other than those shown in the previous table are not supported. The LB_GETSELAREA and LB_SETSELAREA messages are new for TV list-box controls.

List-Box Notification Codes

The following table lists the notification codes applications can receive from a TV list box:

Notification Code	Description
LBN_SELCHANGE	Notifies the application that the current selection has just changed.
LBN_ERRSPACE	Notifies the application that there is not enough memory to satisfy the last request.
LBN_SETFOCUS	Notifies the application that a list-box control has just received the focus.
LBN_KILLFOCUS	Notifies the application that a list-box control has just lost the focus.

Standard Windows list-box notification codes other than those shown in the previous table are not supported. There are no new notification codes for TV list-box controls.

Owner-Draw TV List Boxes

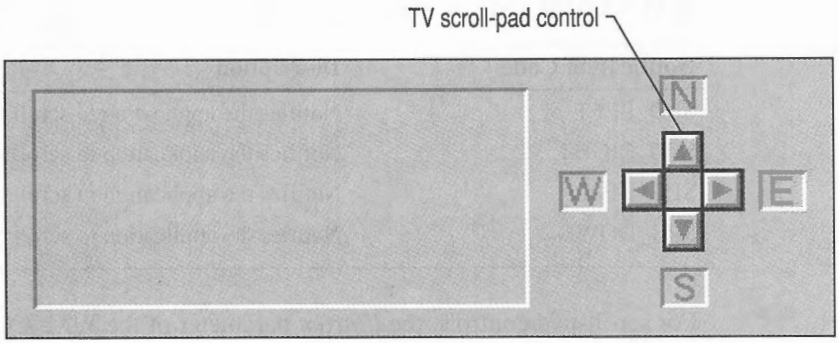
Creating and managing owner-draw TV list boxes is a process similar to creating and managing standard Windows owner-draw list boxes.

▶ **To create and manage owner-draw list boxes:**

1. Use the LBS_OWNERDRAW style when creating the list box.
2. Report the size of the items in the list box when the WM_MEASUREITEM message is received.
3. Draw the appropriate list-box element when the WM_DRAWITEM message is received.

TV Scroll-Pad Control (TVSCROLLPAD)

The scroll-pad control is a new control that allows both horizontal and vertical scrolling by using a single control. The following illustration shows the appearance of the TV scroll-pad control along with horizontal and vertical gauge controls:



TV scroll-pad control

The scroll pad has four focus areas indicating up, down, left, and right scroll directions. Pressing the direction buttons on the hand control moves the focus between these four areas. Pressing the action button initiates a scroll event in the indicated direction.

TV Scroll-Pad Styles

There are two control styles for scroll-pad controls:

Style	Description
SPDS_VERT	Creates a scroll pad with only the vertical scroll arrows enabled.
SPDS_HORZ	Creates a scroll pad with only the horizontal scroll arrows enabled.

Applications should specify both the SPDS_VERT and SPDS_HORZ styles to create a scroll pad with all four scroll arrows enabled.

Getting Input from a TV Scroll Pad

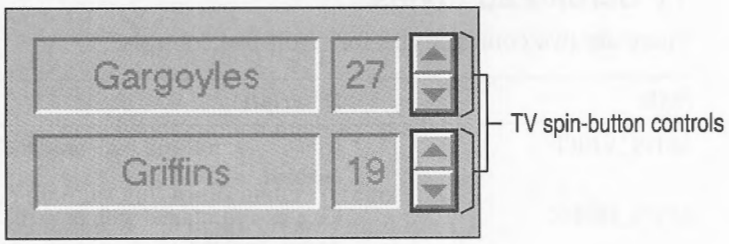
When the user presses the action button while the focus is on a scroll-pad control, the control sends its parent window a WM_COMMAND message to indicate the scroll direction. The *wParam* parameter contains the ID of the control and the *lParam* parameter contains a notification code specifying the scroll direction. The following table lists the scroll-pad notification codes associated with the WM_COMMAND message:

Notification Code	Description
SPD_LEFT	Notifies the application to scroll left.
SPD_RIGHT	Notifies the application to scroll right.
SPD_UP	Notifies the application to scroll up.
SPD_DOWN	Notifies the application to scroll down.

For scroll-pad controls, the *lParam* parameter of the WM_COMMAND message is a bit field—two notification codes can be set in a single WM_COMMAND message to indicate diagonal movement.

TV Spin-Button Control (TVSPINBUTTON)

The TV spin-button control is a new control that allows users to indicate an incremental value or position change. The following illustration shows the appearance of TV spin-button controls:



TV spin-button controls

Generally, spin-button controls are used as a component of another control, such as a scroll bar.

TV Spin-Button Styles

There are two control styles for spin-button controls:

Style	Description
SPINBS_VERT	Creates a vertical spin-button control.
SPINBS_HORZ	Creates a horizontal spin-button control.

Getting Input from TV Spin Buttons

Getting input from spin buttons is similar to getting input from scroll bars. Spin buttons send applications WM_VSCROLL and WM_HSCROLL messages with SB_LINEUP and SB_LINEDOWN identifiers in the *wParam* parameter.

TV Static Control (TVSTATIC)

The TV static control provides simple text fields, icons, and rectangles that can be used to label, box, group, or separate other controls. The TV static control is not an active control—it does not accept input nor provide any output. It is identical to the standard Windows static control, with the following exceptions:

- The appearance of a TV static control does not change when it is disabled.
- TV static controls ignore the SS_NOPREFIX style bit because keyboard accelerators are not supported.

TV Static-Control Styles

The following table lists the control styles for TV static controls:

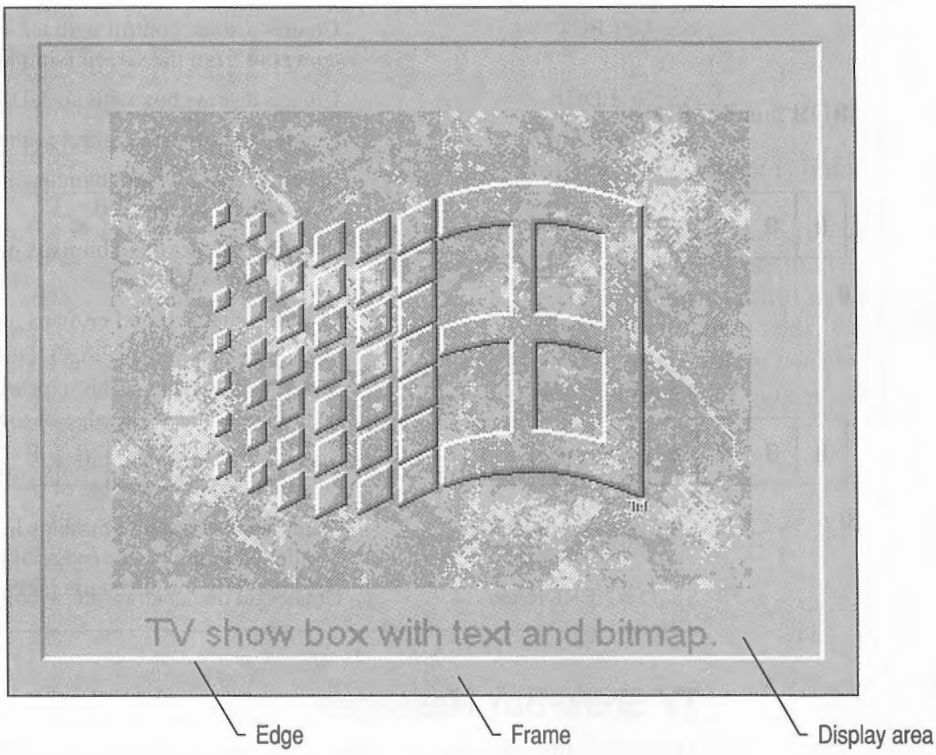
Style	Description
SS_BLACKFRAME	Specifies a box with a frame drawn in the same color as window frames.
SS_CENTER	Designates a simple rectangle and displays the given text, centered in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line automatically wrap to the beginning of the next centered line.
SS_GRAYFRAME	Specifies a box with a frame drawn with the same color as the screen background (desktop). This color is light gray in the default Windows 3.1 and Modular Windows color schemes.

Continued

Style	Description
SS_GRAYRECT	Specifies a rectangle filled with the color used to fill the screen background. This color is light gray in the default Windows 3.1 and Modular Windows color schemes.
SS_ICON	Designates an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. The *nWidth* and *nHeight* parameters are ignored; the icon automatically sizes itself.
SS_LEFT	Designates a simple rectangle and displays the given text, left-aligned in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line automatically wrap to the beginning of the next line.
SS_LEFTNOWORDWRAP	Designates a simple rectangle and displays the given text left-aligned in the rectangle. Tabs are expanded but words are not wrapped. Text that extends past the end of a line is clipped.
SS_RIGHT	Designates a simple rectangle and displays the given text, right-aligned in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line automatically wrap to the beginning of the next line.
SS_SIMPLE	Designates a simple rectangle and displays a single line of text left-aligned in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the WM_CTLCOLOR message.)
SS_WHITEFRAME	Specifies a box with a frame drawn in the same color as the window background. This color is white in the default Windows 3.1 color scheme and light gray in the default Modular Windows color scheme.
SS_WHITERECT	Specifies a rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows 3.1 color scheme and light gray in the default Modular Windows color scheme.
SS_BLACKRECT	Specifies a rectangle filled with the color used to draw window frames. This color is black in the default Windows 3.1 and Modular Windows color schemes.

TV Show-Box Control (TVSHOWBOX)

The TV show-box control provides a single line of text and a bitmap within a 3-D frame. It is not an active control—it does not accept input nor provide any output. The following illustration shows a TV show-box control containing a bitmap and text:



TV show-box control

As shown in the previous illustration, TV show-box controls have three components: a frame, a 3-D edge, and a display area. The frame and edge areas are optional, depending on the style given when the show box is created.

TV Show-Box Control Styles

The following table lists the control styles for TV show-box controls:

Style	Description
SS_DOWNRECT	Creates a static control with a 3-D edge that appears to recede into the screen.
SS_UPRECT	Creates a static control with a 3-D edge that appears to come out from the screen (similar to a push button).
SS_NOEDGE	Creates a show box with no 3-D edge area.
SS_NOFRAME	Creates a show-box control without a frame area.
SS_HCENTER	Specifies that text and bitmaps in the show box should be horizontally centered.
SS_VCENTER	Specifies that text and bitmaps in the show box should be vertically centered.
SS_LEFTALIGN	Specifies that text and bitmaps in the show box should be aligned to the left edge of the control.
SS_RIGHTALIGN	Specifies that text and bitmaps in the show box should be aligned to the right edge of the control.
SS_TOPALIGN	Specifies that text and bitmaps in the show box should be aligned to the top edge of the control.
SS_BOTTOMALIGN	Specifies that text and bitmaps in the show box should be aligned to the bottom edge of the control.
SS_OWNERDRAW	Creates an owner-draw show box.

TV Show-Box Messages

The following table lists messages applications can send to TV show-box controls:

Style	Description
SM_GETDISPLAYEXTENT	Retrieves the display rectangle of a show-box control.
SM_SETDISPLAYEXTENT	Sets the display rectangle of a show-box control.

TV Keyboard Control (TVKEYBOARD)

The TV keyboard control allows users to enter text using a hand control. The TV keyboard control has two basic styles: with and without a prompt and text display area. The following table lists the control styles for TV keyboard controls:

Style	Description
KS_DONOTWRAP	Specifies that the soft-keyboard control allow focus changes to other controls. This style should not be combined with the KS_SYSMODAL style.
KS_SYSMODAL	Specifies that the soft-keyboard control function like a system-modal dialog box. With this style, the focus cannot be changed to other windows until the soft-keyboard control is destroyed.
KS_WITHTEXT	Specifies that the soft-keyboard control have a prompt and text display area.

TV Keyboard Control with Prompt and Text Display

The TV keyboard control with a prompt and text display area allows a user to edit a single line of text and then choose the Enter or Cancel button to notify the application. The application can provide a line of text prompting or querying the user. When notified, the application can then retrieve the text from the keyboard control and either destroy the control or reactivate it and provide another prompt. The following illustration shows the appearance of a TV keyboard control with a prompt and text-display area:

TV keyboard control with prompt and text display (KS_WITHTEXT style)

The default style of the TV keyboard control does not have a prompt and text-display area—to create a TV keyboard control with a prompt and text-display area, you must specify the KS_WITHTEXT style when creating the control. TV keyboard controls are not scalable—**CreateWindow** ignores the *nWidth* and *nHeight* parameters when creating a TV keyboard control.

TV Keyboard (KS_WITHTEXT Style) Messages

The following table lists messages used by TV keyboards with a prompt and text-display area:

Message	Description
KM_GETDEFKEY	Sent by an application to retrieve the state of the Caps Lock button and the button with the input focus.
KM_GETPROMPT	Sent by an application to retrieve the contents of the prompt.
KM_GETPROMPTLENGTH	Sent by an application to retrieve the number of characters in the prompt.
KM_GETTEXTLIMIT	Sent by an application to retrieve the number of characters the user can enter.
KM_MOVESKB	Changes the location of a TV keyboard control.
KM_SETDEFKEY	Sets the state of the Caps Lock button and the button with the initial focus.
KM_SETDEFTEXT	Sets the text appearing in the text area of a TV keyboard control.
KM_SETPROMPT	Sets the text appearing in the prompt area of a TV keyboard control.
KM_SETRECIPIENT	Sent by an application to specify which window is to be notified when the user chooses the Enter or Cancel button.
KM_WAKEUP	Re-enables a TV keyboard control after the user chooses Enter or Cancel.

Getting Input from a TV Keyboard (KS_WITHTEXT Style)

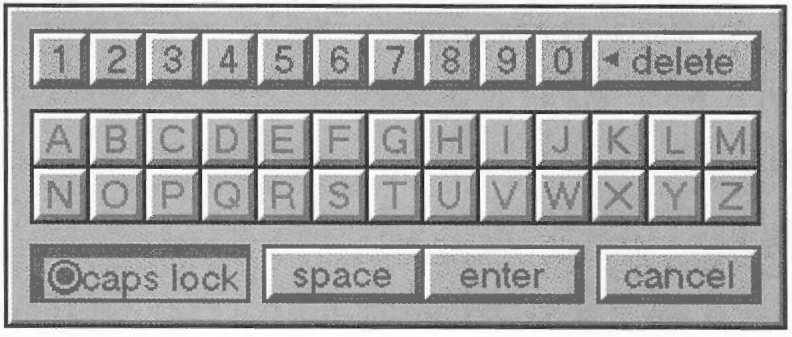
Applications must send a KM_SETRECIPIENT message to a TV keyboard control before they can receive any input from the control. This message tells the keyboard control which window should receive the input. The application does not receive any input as the user "types" on the keyboard—it is only when the user chooses the Enter or Cancel button that the application is notified. Notification is done using a WM_COMMAND message containing either an EN_ENTER or EN_CANCEL notification in the high-order word of the *lParam* parameter. Applications can then use the **GetWindowText** function to get the text entered by the user.

After the TV keyboard control notifies the application of input, it remains on the screen in an inactive state unless it is explicitly destroyed by the application. This allows the application to prompt for additional input by changing the prompt and reawakening the keyboard control. After receiving EN_ENTER or EN_CANCEL notification, applications must either destroy the keyboard or reawaken it. The following code fragment processes a WM_COMMAND message by destroying the keyboard control if EN_CANCEL notification is received, and reawakening it for additional input if EN_ENTER notification is received:

```
case WM_COMMAND:
    switch (wParam)
    {
    case IDD_KEYBOARD:
        // Destroy the keyboard control on Cancel button
        if (HIWORD(lParam) == EN_CANCEL)
            PostMessage(hwndKeyboard, WM_CLOSE, 0, 0L);
        // Get text and reawaken the keyboard on Enter button
        else if (HIWORD(lParam) == EN_ENTER) {
            GetWindowText(hwndKeyboard, textBuffer, 128);
            PostMessage(hwndKeyboard, KM_WAKEUP, 0, 0L);
        }
        break;
        .
        .
        .
    }
    break;
```

Basic TV Keyboard Control

The basic TV keyboard control does not have a text display area—characters are sent to the application as they are entered. The application is responsible for displaying the text as it is entered. The following illustration shows the appearance of the basic TV keyboard control:



Basic TV keyboard control

Basic TV Keyboard Messages

The following table lists messages used by basic TV keyboards:

Message	Description
KM_GETDEFKEY	Sent by an application to retrieve the state of the Caps Lock button and the button with the focus.
KM_MOVESKB	Sent by an application to change the location of a TV keyboard control.
KM_SETDEFKEY	Sent by an application to set the state of the Caps Lock button and the button with the initial focus.
KM_SETRECIPIENT	Sent by an application to specify the window where a TV keyboard should send KM_KEYUP, KM_KEYDOWN, and KM_CHAR messages.
KM_KEYDOWN	Notifies the application that a button on a TV keyboard has been pressed.
KM_KEYUP	Notifies the application that a button on a TV keyboard has been released.
KM_CHAR	Notifies the application that a button on a TV keyboard has been pressed and released.

Getting Input from a Basic TV Keyboard

Applications must send a KM_SETRECIPIENT message to a TV keyboard control before they can receive any input from the control. This message tells the keyboard control which window should receive the input. Then, the TV keyboard control notifies the application of input by using the KM_KEYDOWN, KM_KEYUP, and KM_CHAR messages.

Unlike TV keyboard controls with a prompt and text display (KS_WITHTEXT style), basic TV keyboard controls don't notify the application when the user chooses the Enter or Cancel buttons. Applications must look for KM_KEYUP messages with a VK_ENTER or VK_CANCEL key code to detect when the user is finished entering text.

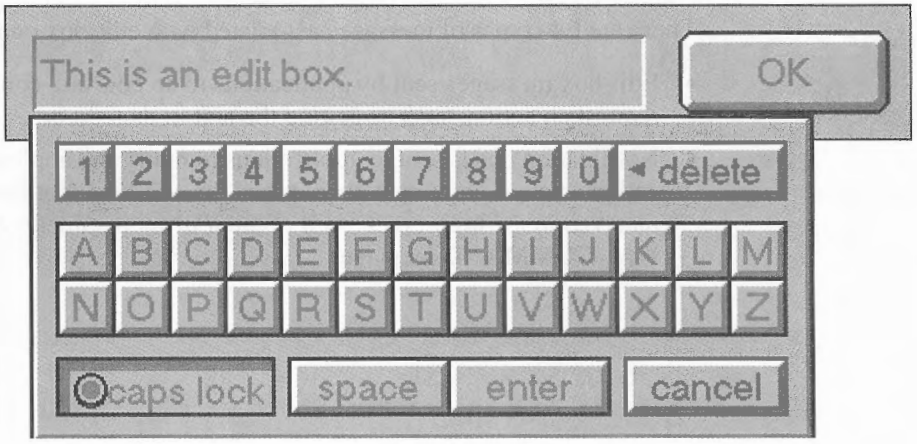
TV Keyboard Controls and Input Focus

TV keyboard controls created using the KS_SYSMODAL style function like modal dialog boxes. When activated, they capture the mouse and all input from the hand control. They don't release the mouse capture or the input focus until they are destroyed. TV keyboard controls created without the KS_SYSMODAL style allow focus changes to other controls. If tabbing mode is in effect, focus changes to other controls can only occur if tabbing is unconstrained. See "Constrained and Unconstrained Tabbing," later in this chapter for information about unconstrained tabbing mode.

TV Edit-Box Control (TVEDITBOX)

The TV edit-box control appears as a single line of text and, when activated, displays a TV keyboard control to allow the user to enter and edit text. The edit-box control is activated when the action button on a hand control is pressed while the edit-box control has the focus. The following illustrations show an edit-box control before and after it is activated:

TV edit-box control before activation



TV edit-box control after activation

To retrieve the text from an edit box, use the **GetWindowText** function or send the control an EM_GETLINE message.

Edit-Box Control Styles

The following table lists the control styles for TV edit-box controls:

Style	Description
ES_PASSWORD	Creates a password edit control that displays all characters as an asterisk (*) as they are entered.
ES_READONLY	Creates an edit box that cannot be edited by the user. The EM_SETREADONLY message can be used to set and clear the read-only flag of an edit-box control.
KS_SYSMODAL	Specifies that the associated soft-keyboard control functions like a system-modal dialog box. With this style, the focus cannot be changed to other windows until the soft-keyboard control is destroyed.
KS_WITHTEXT	Specifies that the associated soft-keyboard control has a prompt and text display area.

The KS_SYSMODAL and KS_WITHTEXT styles determine the style of the TV keyboard control displayed when the edit control is activated. Edit controls with the KS_WITHTEXT style set display a TV keyboard control with a prompt and text-display area. The text in the edit-box control is not modified until the user chooses the Enter button. If the user chooses the Cancel button, the text in the edit box is not changed.

TV Edit-Box Control Messages

There are two types of messages associated with edit-box controls:

- Edit-box messages sent by applications to an edit-box control to initiate or request an action, such as getting the text in the edit box.
- Notification codes sent to applications to notify the application of a user action, such as entering a character in the edit box. Notification codes are received in the high-order word of the *lParam* parameter of a WM_COMMAND message.

Edit-Box Messages

The following table lists messages applications can send to TV edit-box controls:

Message	Description
EM_GETLINE	Sent by an application to retrieve the text from an edit-box control.
EM_GETMODIFY	Sent by an application to query if the text in the edit-box control has been modified.
EM_GETRECT	Sent by an application to get the rectangle of the text area of the edit-box control.
EM_LINELENGTH	Sent by an application to query the length (in bytes) of an edit-box text buffer.
EM_SETREADONLY	Sent by an application to set the read-only flag for an edit-box control.
EM_SETMODIFY	Sent by an application to set the modify flag for an edit-box control.
EM_SETPASSWORDCHAR	Sent by an application to set the character displayed in an edit-box control created using the ES_PASSWORD style. The default display character is an asterisk (*).
KM_GETDEFKEY	Sent by an application to retrieve the state of the Caps Lock button and the button with the input focus on the keyboard control associated with an edit box.
KM_GETPROMPT	Sent by an application to retrieve the contents of the prompt on the keyboard control associated with an edit box.
KM_GETPROMPTLENGTH	Sent by an application to retrieve the number of characters in the prompt on the keyboard control associated with an edit box.
KM_GETTEXTLIMIT	Sent by an application to query the maximum number of characters a user is allowed to enter in the keyboard control associated with an edit box.
KM_MOVESKB	Sent by an application to change the location of the keyboard control associated with an edit box.
KM_SETDEFKEY	Sent by an application to set the state of the Caps Lock button and the button with the initial focus in the keyboard control associated with an edit box.
KM_SETDEFTEXT	Sent by an application to set the text appearing in the text area of the keyboard control associated with an edit box.
KM_SETPROMPT	Sent by an application to set the text appearing in the prompt area of the keyboard control associated with an edit box.

Edit-Box Notification Codes

The following table lists the notification codes applications can receive from a TV edit-box control:

Notification Code	Description
EN_CHANGE	Notifies the application that the user has changed the text in the edit-box control. This message is sent after the display is updated.
EN_ERRSPACE	Notifies the application that the edit-box control is out of memory.
EN_KILLFOCUS	Notifies the application that the edit-box control is losing the input focus.
EN_MAXTEXT	Notifies the application that the edit-box buffer is full and text has been truncated.
EN_SETFOCUS	Notifies the application that the edit-box control is receiving the input focus.
EN_UPDATE	Notifies the application that the user has changed the text in the edit-box control. This message is sent before the display is updated so that the application can resize the edit box.

Predefined Control-Class Names

The following table lists the predefined control-class names for the TV user-interface controls. Applications can use these classes with the **CreateWindow** function to create child-window controls.

Control-Class Name	Description
TVBUTTON	Push button, radio button, check box, and group box
TVEDITBOX	Edit-box control
TVKEYBOARD	TV keyboard
TVLISTBOX	List box
TVSCROLLBAR	Scroll bar and gauge
TVSCROLLPAD	Scroll pad
TVSHOWBOX	Show box
TVSPINBUTTON	Spin button
TVSTATIC	Static control

TV User-Interface Functions

The following functions control settings that affect all TV user-interface controls, such as how focus is indicated and how controls are drawn on the screen:

tvGetStockObject
Retrieves a handle to one of the predefined stock objects for user-interface controls (SMALLFONT or BIGFONT).

tvSetHighlightFrame
Sets the highlight color and blinking rate used to indicate focus.

tvGetHighlightFrame
Retrieves the current highlight color and blinking rate used to indicate focus.

tvSetUIFlags
Allows applications to change internal TV user-interface flags. These flags affect how controls are drawn and how the cursor moves on focus changes.

tvGetUIFlags
Retrieves the internal TV user-interface flags set using **tvSetUIFlags**.

For information about these functions, see Chapter 5, "Function Directory."

Adding Bitmaps to Controls

Some of the TV user-interface controls can be customized with bitmaps supplied by applications. The following table lists the controls that can be customized:

Control	Customization
Push button, check box, and radio button	Add a bitmap to the face of button or supply a bitmap for the entire button.
Group box	Add a bitmap to the face of the group box.
Scroll bar and gauge	Supply a bitmap for the thumb.
Show box	Add a bitmap to the face of the control.

Messages for Getting and Setting Bitmaps in Controls

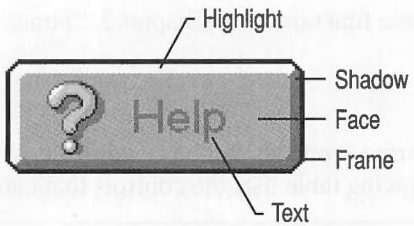
The following table lists messages used to retrieve and set bitmaps used in controls:

Message	Description
WM_SETBITMAP	Supplies a bitmap for a control.
WM_GETBITMAP	Retrieves a bitmap used by a control.

Applications must create and delete bitmap objects passed to controls. Bitmaps should not be deleted until after the controls using them have been destroyed.

Changing Control Colors

The TV user-interface controls allow applications to change the colors used in rendering the controls. The following illustration identifies the elements of a button control with colors that can be changed:



Elements of a TV button control

Messages for Changing Control Colors

The following table lists messages used to retrieve and set the colors used in controls:

Message	Description
WM_SETCOLOR	Changes the color used to render a control element.
WM_GETCOLOR	Retrieves the color used to render a control element.

Enabling and Disabling Controls

You can use the **EnableWindow** function to enable and disable Modular Windows user-interface controls. Disabled controls can't receive the focus and will not respond to user input. Disabled controls are drawn without their 3-D appearance to distinguish them from enabled controls.

Note Don't disable a control if the control has the input focus—change the focus to another control before disabling the control.

The Focus Manager

In tabbing mode, the focus manager is responsible for moving the focus between controls when the user presses the direction buttons on a hand control. The TV user-interface controls are automatically handled by the focus manager, although applications can override the focus manager for individual controls or for all controls.

Applications that create custom controls can use the focus manager to handle focus changes between controls. Only applications using custom controls or applications that override the actions of the focus manager need to make calls to the focus-manager functions.

Note Controls must have the WS_TABSTOP style set to receive the focus.

Focus-Manager Functions

The following functions allow applications to interact with the focus manager:

fmAddWindow
Adds a control to the list of controls managed by the focus manager.

fmDeleteWindow
Deletes a control from the list of controls managed by the focus manager.

fmSetWindowVectors
Sets the focus vectors for a control. Focus vectors determine the next control that gets the focus when the focus is moved.

fmGetWindowVectors
Gets the focus vectors for a control.

fmGetLastCursorPos
Retrieves the last cursor position set by **fmSetCursorPos**.

fmSetCursorPos
Sets the position of the cursor indicating which control has the focus.

fmIsFocusMessage
Sends WM_KEYDOWN messages to the focus manager for processing.

fmGetLastDirection
Gets the direction of the last focus movement.

fmTranslateHCKey
Selectively translates WM_KEYDOWN and WM_KEYUP messages generated by the hand control into unmapped virtual key codes.

For detailed information about these functions, see Chapter 5, "Function Directory."

Focus-Manager Messages

Modular Windows provides a focus-manager message, WM_QUERYFOCUS, to query which element of a compound control has the focus. For example, you can use this message to determine which scroll arrow in a scroll-bar control has the focus. For detailed information about this message, see Chapter 6, "Message Directory."

Constrained and Unconstrained Tabbing

In tabbing mode, the focus manager handles focus changes between controls when the user presses direction buttons on the hand control. Tabbing can either be *constrained* or *unconstrained*, which is defined as follows:

- Constrained tabbing limits focus changes to controls with the same parent window as the control currently with the focus.
- Unconstrained tabbing allows focus changes to controls with different parent windows.

By default, tabbing is constrained. To enable unconstrained tabbing, applications must call the **tvSetUIFlags** function to clear the TVFMCHECKPARENT flag.

Using the Focus Manager with Custom Controls

To use the focus manager with custom controls, applications must follow these steps:

1. Add controls to the focus manager (**fmAddWindow**).
2. Set the direction vectors for focus changes (**fmSetWindowVectors**). This step is optional—the focus manager provides a default focus-movement algorithm.
3. Pass WM_KEYDOWN messages to the focus manager (**fmIsFocusMessage**).

Adding Controls to the Focus Manager

Any custom control can choose to use the focus manager to handle focus changes. To add a control to the focus manager, the application should call the **fmAddWindow** function when the window procedure for the custom control receives a WM_CREATE message.

Setting Focus-Direction Vectors

The focus manager provides a default focus-movement algorithm which should be sufficient for all but the most complex set of controls. Applications can provide their own focus-direction vectors for any controls managed by the focus manager.

To set the focus-direction vectors for a control, applications pass a **DIRVECTORS** structure to the focus manager using the **fmSetWindowVectors** function. The **DIRVECTORS** structure specifies the control that gets the focus when the focus is moved up, down, left, or right. The **DIRVECTORS** structure uses window handles to identify controls.

Passing WM_KEYDOWN Messages to the Focus Manager

The window procedure for controls that have been added to the focus manager using **fmAddWindow** must pass WM_KEYDOWN messages to the focus manager so that the focus manager can determine if the key event is from a hand-control button, which changes the focus.

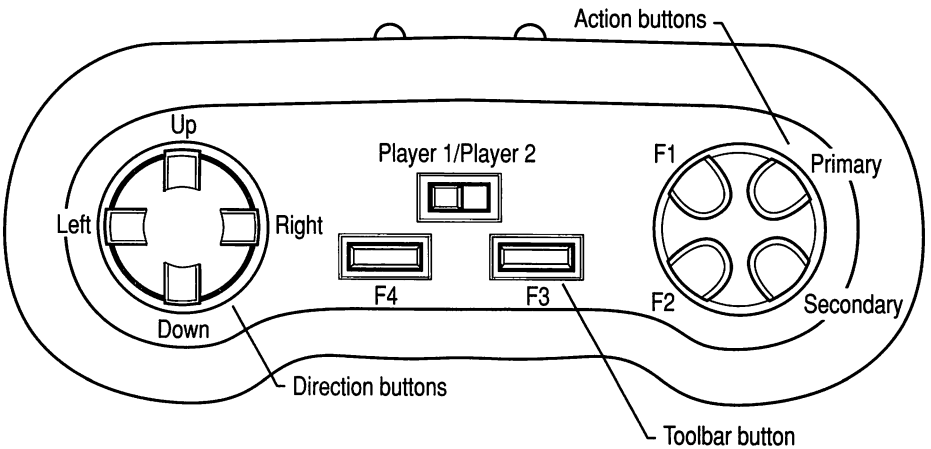
CHAPTER 2

Hand-Control Services

This chapter describes the Modular Windows hand-control services and shows how applications can use these services to get input from the hand control. For information about user-interface design for the hand control, see Chapter 1, "User-Interface Controls."

The Hand Control

With TV-based multimedia players, the primary input device is a remote hand control—a mouse and keyboard are optional. The hand control has ten buttons. Four of these buttons make up a direction pad, two buttons are the primary and secondary action buttons, and the remaining four buttons are function buttons. The following illustration shows the layout of the ten buttons on a hand control:



Hand control for TV-based multimedia players

Hand-Control Functions and Macros

The following functions and macros allow applications to program the hand-control driver, get and set the cursor position, and work with the virtual key codes generated by the hand-control driver:

hcControl
Changes hand-control settings such as the virtual key mapping, cursor-movement parameters, and the user-interface mode.

hcGetCursorPos
Gets the current cursor position.

hcSetCursorPos
Sets the current cursor position.

HC_IS_HC
Determines if a given virtual-key event is from the hand control.

HC_KEY_OFFSET
Calculates the offset in the hand-control key map for a given hand-control button.

HC_PLAYER
Determines if a given hand-control key event is from the first hand control or the second hand control.

HC_VKN2VK
Converts a virtual key code for the second hand control to the equivalent virtual-key code for the first hand control.

For detailed information about these functions and macros, see Chapter 5, "Function Directory."

Getting Input from the Hand Control

When a user presses or releases a button on the hand control, the hand-control driver sends WM_KEYDOWN, WM_KEYUP, and WM_CHAR messages to the window procedure for the window with the input focus. The HC.H header file defines the following constants for the virtual-key codes generated by the hand-control driver:

Constant	Description
VK_HC1_UP	Up direction-control button
VK_HC1_DOWN	Down direction-control button
VK_HC1_LEFT	Left direction-control button
VK_HC1_RIGHT	Right direction-control button
VK_HC1_PRIMARY	Primary action button
VK_HC1_SECONDARY	Secondary action button
VK_HC1_F1	Function button 1
VK_HC1_F2	Function button 2
VK_HC1_F3 (VK_HC1_TOOLBAR)	Function button 3 (toolbar button)
VK_HC1_F4	Function button 4

The HC.H header file defines a similar set of constants (VK_HC2_UP, etc.) for virtual-key codes generated by the buttons on a hand control, while set to "Player 2" mode.

Note Hand-control key events are tagged with an extra DWORD of information accessible with the **GetMessageExtraInfo** function. This DWORD is set to the four-character code "HKEY" (type FOURCC). Use the HC_IS_HC macro to determine if a key event is from the hand control.

Tabbing and Roaming Modes

The **hcControl** function allows applications to choose how each hand control (player 1 and player 2) affects focus and cursor movement. Two input modes are available: tabbing and roaming modes.

Using Tabbing Mode

Tabbing mode is the preferred mode for applications using the TV user-interface controls. In tabbing mode, the user presses the direction buttons on a hand control to discretely move the focus between controls. See Chapter 1, "User-Interface Controls," for details on using tabbing mode.

Tabbing mode can be disabled for either player 1 or for player 2 by calling the **tvSetUIFlags** function and setting the TVFMNOTAB1 or TVFMNOTAB2 bits in the *wNewFlags* parameter. Two-player applications might want to use this feature to allow one hand control to be the master control with access to user-interface controls.

Using Roaming Mode

In roaming mode, the hand control operates similarly to a mouse; the user presses the direction buttons to iteratively move the cursor to the desired location. The primary action button can be remapped to simulate a mouse button, as shown in the following example code fragment:

```
hcControl(HC_SET_ROAM, 1, (LPARAM)(HC_ROAM1 | HC_ROAM2));
hcControl(HC_SET_KEY, HC_KEY_OFFSET(VK_HC1_PRIMARY), VK_LBUTTON);
```

CHAPTER 3

Video Services

This chapter explains how to use the Modular Windows display drivers, display images using the **DisplayDib** function, and directly access video memory.

Display Drivers

Modular Windows includes a display driver, TVVGA.DRV, a dual-mode 256-color palettized driver that can operate in either high-resolution (640-by-400) or low-resolution (320-by-200) mode. The TVVGA.DRV driver has the following enhancements to improve performance with televisions:

- The width of lines used to draw window borders is increased from one to two pixels to reduce interlace flicker.
- Dithering is disabled, which also reduces interlace flicker.
- The intensity of the 20 reserved system colors is reduced to prevent color smearing.
- The size of the default cursor (the arrow cursor) is enlarged for better visibility with the extended viewing range of televisions.

Choosing Display Driver Resolution

The TVVGA.DRV display driver can operate in either high-resolution (640-by-400) or low-resolution (320-by-200) mode. The default mode is high resolution. To use the low-resolution mode, add the following section to the SYSTEM.INI file:

```
[TVVGA]
resolution=320x200x8
```

Use the following guidelines to help choose which display driver resolution to use with your application:

- Choose high-resolution mode for applications that use the TV user-interface controls. The TV user-interface controls are optimized for high-resolution mode.
- Choose high-resolution mode for applications that present large amounts of text. Text is more legible in high-resolution mode.
- Choose low-resolution mode for applications that don't use the TV user-interface controls, but present a custom user interface using bitmaps, such as an arcade game. Because low-resolution mode is not interlaced, interlace-flicker artifacts are eliminated. Also, performance is better in low-resolution mode because the size of each bitmap is smaller by a factor of 4.

Using the NEWTRANSPARENT Background Mode

The TVVGA.DRV display driver supports a new background mode called NEWTRANSPARENT mode. This background mode allows bitmaps to have transparent areas—when the bitmaps are drawn to the screen, the pixels corresponding to the transparent areas remain unchanged.

▶ **To use NEWTRANSPARENT mode to draw bitmaps with transparent areas:**

1. Choose one of the default palette colors to represent transparency.
2. Create bitmaps using the chosen transparent color in the areas you want to be transparent.
3. Before drawing the bitmap to the display, set the background mode to NEWTRANSPARENT and the background color to the transparent color.

The following code fragment sets the background mode to NEWTRANSPARENT and the transparent color to 100% green:

```
// Set background mode and color
SetBkMode(hdc, NEWTRANSPARENT);
SetBkColor(hdc, PALETTERGB(0,255,0));

// Blit bitmap
.
.
.
```

To ensure that NEWTRANSPARENT mode works properly with the Modular Windows default palette, use the **PALETTERGB** macro to specify the transparent color in the **SetBkColor** function.

About the Default Palette

Modular Windows reserves 20 static colors in the system palette. These 20 colors, called the *default palette*, are reserved and cannot be changed by applications. The colors are used to draw icons and 16-color bitmaps, and to draw elements of the Modular Windows user interface, such as window borders and controls. Modular Windows uses the same colors in the default palette as Microsoft Windows 3.1, however, to reduce the artifact of color smearing on televisions, the intensity of some of the colors in the Modular Windows default palette is reduced.

Note Applications should not depend on the existence of specific RGB values in the default palette.

Avoiding Color Matching Anomalies

The color tables of icons, 16-color bitmaps, and 256-color bitmaps with identity palettes use RGB values from the Windows 3.1 default palette. When displayed on Modular Windows, some of the RGB values of the low-intensity colors in the color table will map to different default-palette indices. The result is that some of the colors might appear to be incorrect. The Modular Windows SDK provides a tool, Color Table Converter, you can use to convert the color table of bitmaps and icons so that they are properly displayed under Modular Windows. See Chapter 9, "Tools," for details on using the Color Table Converter tool.

For more information about using palettes and avoiding color-matching problems, see the "Video Techniques" chapter in the *Programmer's Reference, Volume 1: Overview* manual in the Microsoft Windows SDK.

The DisplayDib and DisplayDibEx Functions

Modular Windows includes the following new versions of the **DisplayDib** function originally included in the Microsoft Multimedia Development Kit:

DisplayDib
Displays a bitmap using the full display area. **DisplayDib** either centers the image or places it in the lower-left corner of the display.

DisplayDibEx
Identical to **DisplayDib** except that it allows applications to specify the screen location for placing the bitmap.

For detailed information about these functions, see Chapter 5, "Function Directory."

Supported File Formats and Resolutions

The **DisplayDib** and **DisplayDibEx** functions can display images in the following file formats:

- 8-bit palettized DIB
- RGB555 16-bit DIB
- RGB565 16-bit DIB
- YUV16
- YUV8

DisplayDib and **DisplayDibEx** display each of these image formats, except 8-bit palettized DIBs, in 320-by-200 resolution and in 320-by-400 resolution. Images in 8-bit palettized DIB format can only be displayed in 320-by-200 resolution. See Chapter 8, "File Formats," for details on these file formats.

Note The 8-bit DIB format is supported by all TV-based players. RGB and YUV formats are supported on VIS players and might be supported on future TV-based players. The **DisplayDib** function allows applications to query to determine which image formats **DisplayDib** supports.

TV-Based Player Pixel Aspect Ratios

The following table lists the pixel aspect ratios for TV-based players:

Resolution	Pixel Aspect Ratio (vertical:horizontal)
640-by-400	1.2 : 1
320-by-400	0.6 : 1
320-by-200	1.2 : 1

The 320-by-400 resolution is only available by using the **DisplayDib** and **DisplayDibEx** functions.

Directly Accessing Video Memory

You might want to directly access the video memory on a TV-based player to do the following:

- To use video modes not supported by Modular Windows. You can use the **EnterDVA** function to disable Windows graphics services and take control of the display. This allows you to change to any video mode supported by a TV-based player.

  This approach is called direct video access (DVA).

- To provide special video effects that cannot be accomplished using Modular Windows GDI functions. You can use special programming techniques to supplement the Windows graphics services by directly accessing the video memory corresponding to the portion of the display within a window. You cannot change the hardware video mode while using this approach.

  This approach is called direct window access (DWA).

Note In deciding to use either method of directly accessing video memory, you must introduce device dependence to your application—porting the application to a multimedia PC or to another TV-based player will be more difficult. If your application must use an unsupported video mode, you should consider writing a device driver for Windows for that mode. Porting the device driver to a new hardware platform might be easier than porting your application.

Direct-Video Access Macros

Modular Windows includes the following macros to support direct-video access:

Macro	Description
EnterDVA	Begins direct-video access.
LeaveDVA	Ends direct-video access.

CHAPTER 4

Core API and Extension Libraries Support

Modular Windows supports many of the functions in the Windows 3.1 API and many MS-DOS® functions. This chapter describes the differences between the Modular Windows API and the Windows 3.1 API, and describes the MS-DOS functions supported by Modular Windows.

Core API Support

The Windows 3.1 core API is contained in three modules: KERNEL, GDI, and USER. Many of these functions are fully supported in Modular Windows. Others are partially supported; for example, the WS_MAXIMIZEBOX window style is not available in the **CreateWindow** function. Some functions from Windows 3.1 are not supported in Modular Windows. For example, functions that result in a write operation to a disk are not supported because TV-based players don't presently support writable file devices.

The Modular Windows SDK provides two include files to help you locate functions not supported in Modular Windows, or those supported with reduced functionality. The include files are UNSUPP.H and CHANGED.H. The compiler will generate warnings when it encounters changed or unsupported functions, and errors when it encounters undefined constants, such as constants for messages that are not supported.

The following is an alphabetical list of functions in the Windows 3.1 core API—the KERNEL, GDI, and USER modules—that have been changed or are not available in Modular Windows:

AddFontResource
Attempts to load TrueType® fonts will fail with a return value of 0. Bitmap fonts can be loaded.

AppendMenu
Unsupported, returns FALSE. Menus are not supported on TV-based players.

ArrangeIconicWindows
Unsupported because minimized windows are not supported. Returns 0, indicating no icons.

CheckMenuItem
Unsupported. Menus are not supported on TV-based players. Returns -1, indicating the designated menu item does not exist.

CloseSound
Unsupported, returns NULL.

CopyMetaFile
This function fails with NULL. All storage devices in TV-based players are read-only.

CountVoiceNotes
Unsupported, returns 0.

CreateFont, CreateFontIndirect
Attempts to create TrueType fonts, will select an alternate font.

CreateMenu
Unsupported, returns NULL.

CreatePopupMenu
Unsupported because menus are not supported on TV-based players. Returns NULL, indicating menu could not be created.

CreateScalableFontResource
Unsupported, returns 0, indicating the specified TrueType font file could not be created.

CreateWindow, CreateWindowEx
Supports the following new control classes: TVBUTTON, TVEDITBOX, TVKEYBOARD, TVLISTBOX, TVSCROLLBAR, TVSCROLLPAD, TVSHOWBOX, and TVSTATIC.

The COMBOBOX and MDICLIENT classes are not supported—the function will return NULL when you try to create these window classes. The BUTTON, EDIT, LISTBOX, SCROLLBAR, and STATIC control classes will be translated into corresponding Modular Windows control classes, as shown in the following table:

Windows 3.1 Class	Modular Windows Class
BUTTON	TVBUTTON
EDIT	TVEDITBOX
LISTBOX	TVLISTBOX
SCROLLBAR	TVSCROLLBAR
STATIC	TVSTATIC

Some styles of Windows 3.1 controls, such as multiple-column list boxes, are not supported in Modular Windows and will be ignored when the controls are translated into corresponding Modular Windows controls.

CreateWindow and **CreateWindowEx** ignore the following window styles:

- WS_HSCROLL
- WS_ICONIC
- WS_MAXIMIZE
- WS_MAXIMIZEBOX
- WS_MINIMIZE
- WS_MINIMIZEBOX
- WS_SCROLL
- WS_SYSMENU

Modular Windows does not support sizing borders on windows—windows created with the WS_THICKFRAME style will not include the lines which visually separate areas where horizontal, vertical, or diagonal sizing is usually performed.

DefFrameProc, DefMDIChildProc
Unsupported, returns 0L, indicating the message was not processed.

DeleteMenu
Unsupported, returns FALSE.

DestroyMenu
Unsupported, returns 0.

DlgDirList, DlgDirListComboBox, DlgDirSelect, DlgDirSelectComboBox, DlgDirSelectComboBoxEx
Unsupported, these functions return 0L.

DrawMenuBar
Unsupported, no return value.

EnableMenuItem
Unsupported, returns -1, indicating no menu item exists.

GetAsyncKeyState
This function works correctly for the virtual keys currently mapped to the hand control, as well as for the virtual keys generated from the optional keyboard.

GetClassInfo
This function will return FALSE for the following Microsoft Windows 3.1 control classes: BUTTON, COMBOBOX, EDIT, LISTBOX, MDICLIENT, and SCROLLBAR.

GetDlgItemInt, GetDlgItemText
Supported only for TVSTATIC and TVECHOBOX controls. These are the only Modular Windows controls that support the WM_GETTEXTLENGTH and WM_GETTEXT messages.

GetFontData, GetGlyphOutline
Unsupported, fails with -1 return value.

GetKeyboardState, GetKeyState
This function works correctly for the virtual keys currently mapped to the hand control, as well as for the virtual keys generated from the optional keyboard.

GetKeyNameText
If the optional keyboard is not installed, this function returns 0.

GetMenu
Unsupported, returns NULL, indicating the window has no menu.

GetMenuCheckMarkDimensions
Unsupported, returns 0; no failure return value is defined for this function.

GetMenuItemCount, GetMenuItemID
Unsupported, returns -1, indicating the menu handle is invalid.

GetMenuState
Unsupported, returns -1.

GetMenuString
Unsupported, returns 0, indicating no bytes were copied to the supplied buffer.

GetModuleFileName
Attempts to retrieve the path for executable files in the ROM, will fail.

GetNextDlgGroupItem
This function is supported as documented in the Windows SDK. In Windows 3.1, the Dialog Manager handles the tabbing order of controls in a group. The controls in a group are returned according to the tabbing order. However, in Modular Windows, the focus manager determines the tabbing order. Applications written for Modular Windows should not rely on the tabbing order given by this function.

GetNextDlgTabItem
Unsupported, returns NULL.

GetSubMenu
Unsupported, returns NULL, indicating no popup menu exists.

GetSystemMenu
Unsupported, returns NULL, indicating the system menu handle does not exist.

GetSystemMetrics
When called with the *nIndex* parameter set to SM_CXHSCROLL, SM_CYHSCROLL, SM_CXVSCROLL, or SM_CYVSCROLL, this function returns a value of 1 because Modular Windows does not support non-client scroll bars.

GetTempDrive
If there is no writable file device in the system, **GetTempDrive** returns drive A as the temporary drive.

GetTempFileName
Unsupported on systems with no writable file devices, returns 0.

GetThresholdEvent, GetThresholdStatus
Unsupported; **GetThresholdEvent** returns NULL, **GetThresholdStatus** returns 0.

GetWinDebugInfo
Unsupported, returns 0.

GetWindowPlacement
Only the **rcNormalPosition** of the **WINDOWPLACEMENT** structure is valid.

GetWindowsDirectory
Returns the Windows directory specified in the MODWIN command line at start time.

_hwrite
Returns HFILE_ERROR on systems with no writable file devices.

HiliteMenuItem
Unsupported, returns 0.

InsertMenu
Unsupported, returns 0.

IsMenu
Unsupported, returns 0.

_lcreat
Returns HFILE_ERROR on systems with no writable file devices.

_lopen
Returns HFILE_ERROR if the file is opened for writing on systems with no writable file devices.

_lwrite
Returns HFILE_ERROR.

LoadAccelerators
Unsupported, returns NULL.

LoadMenu, LoadMenuIndirect
Unsupported, returns NULL.

mciGetErrorString
To save space in the Modular Windows ROM, this function fills the specified buffer with the text "MCI Error: XXXXH" where XXXX is the hexadecimal MCI error code. In Windows 3.1, this function fills the buffer with a textual description of the error. The error descriptions returned by **mciGetErrorString** should not be presented to users. See "MCI Error Codes," later in this chapter, for a list of MCI error codes.

MessageBox
Uses Modular Windows TVBUTTON class. Does not include the system menu.

midiOutOpen
Because Modular Windows does not support the MIDI Mapper, this function will fail if MIDI_MAPPER is specified for the *wDeviceID* parameter.

ModifyMenu
Unsupported, returns 0, indicating the menu could not be modified.

NetBIOSCall
Unsupported, return value is undefined.

OpenFile
Supported if a file is opened as read-only. Fails and returns -1 is a file is opened as anything other than read-only on a system that does not include writable file devices.

OpenSound
Unsupported, returns NULL.

RemoveMenu
Unsupported, returns 0, indicating the menu was not removed.

SetMenu, SetMenuItemBitmaps
Unsupported, returns 0.

SetSoundNoise
Unsupported, no return value.

SetTargetDevice
Unsupported, returns an error value.

SetVoiceAccent, SetVoiceEnvelope, SetVoiceNote, SetVoiceQueueSize, SetVoiceSound, SetVoiceThreshold
Unsupported, no return value.

SetWinDebugInfo
Unsupported, returns 0.

SetWindowPlacement
Modular Windows does not support minimized or maximized windows. The **SetWindowPlacement** function only supports settings associated with the normal window position and hiding and showing the window.

ShowWindow
Modular Windows does not support minimized or maximized windows. The **ShowWindow** function does not support values associated with maximized or minimized windows.

StartSound, StopSound
Unsupported, no return value.

SwapRecording
Unsupported, no return value.

SyncAllVoices
Unsupported, no return value.

SystemParametersInfo
On systems with no writable file devices, changes to system parameters will not be saved and will only be valid for the current Modular Windows session.

TrackPopupMenu
Unsupported, returns 0.

TranslateAccelerator
Unsupported, returns 0.

TranslateMDISysAccel
Unsupported, returns 0.

ValidateCodeSegments
Unsupported, not available in ROM.

ValidateFreeSpaces
Unsupported, not available in ROM.

WaitSoundState
Unsupported, no return value.

WritePrivateProfileString, WriteProfileString
Unsupported on systems that don't include writable file devices.

Extension Libraries Support

Some of the Microsoft Windows 3.1 extension libraries are directly supported in Modular Windows. Others are supported with some restrictions. Some libraries, such as the common dialog-box library (COMMDLG.DLL), are not available in Modular Windows.

The following table summarizes Modular Windows support for various Windows extension libraries:

Library	Support
Common Dialog Boxes (COMMDLG.DLL)	No
DDE Management (DDEML.DLL)	No
OLE (OLECLI.DLL, OLESVR.DLL)	No
Registration Database (SHELL.DLL)	Yes, with restrictions
Toolhelp (TOOLHELP.DLL)	Yes
Data Decompression (LZEXPAND.DLL)	Yes, with restrictions
Stress Testing (STRESS.DLL)	Yes, with restrictions
32-Bit Memory Management (WINMEM32.DLL)	No
Floating Point Emulation (WIN87EM.DLL)	Yes

Registration Database (SHELL.DLL)

The registration database stores information that defines associations between applications and files. In Windows 3.1, this information is used by File Manager during drag-and-drop operations, and by OLE server applications during operations involving linked or embedded objects.

Modular Windows does not require the same functionality from a registration database because:

- Modular Windows does not have a File Manager, and so does not require support for file associations.
- The registration database (REG.DAT) cannot be modified on systems without writable file devices.

Provided that SHELL.DLL, WIN.INI, and REG.DAT are in the Microsoft Windows 3.1 directory, the following functions operate the same as under Windows 3.1:

DragAcceptFiles
DragFinish
DragQueryFile
DragQueryPoint
FindExecutable
RegCloseKey
RegEnumKey
RegOpenKey
RegQueryKey
RegQueryValue
ShellExecute

The following functions will fail on systems without writable file devices:

RegCreateKey
RegDeleteKey
RegSetValue

Data Decompression (LZEXPAND.DLL)

Functions in LZEXPAND.DLL are supported with the following exceptions:

- **LZCopy** is not supported on systems with no writable file devices.
- **CopyLZFile** is not supported on systems with no writable file devices.
- For **LZOpenFile**, the OF_CREATE, OF_DELETE, OF_WRITE, and OF_READWRITE styles are not supported on systems with no writable file devices.

Stress Testing (STRESS.DLL)

Functions in STRESS.DLL are supported with the following exceptions:

- **AllocDiskSpace** is not supported on systems with no writable file devices.
- **UnAllocDiskSpace** is not supported on systems with no writable file devices.

MS-DOS Function Support

Modular Windows provides MS-DOS functions through the INT 20H and 21H interrupts. Not all of the standard INT 21H functions are available—unsupported functions will return with the carry flag (CY) set.

Unchanged INT 21H Functions

The following INT 21H functions are supported without any changes:

Register AH	Function
00H	Terminate Process
0EH	Set Default Drive
19H	Get Default Drive
1AH	Set Disk Transfer Address
25H	Set Interrupt Vector
26H	Create New PSP
2FH	Get Disk Transfer Address
30H	Get MS-DOS Version Number
31H	Terminate and Stay Resident
34H	Get InDOS Flag Address
35H	Get Interrupt Vector
45H	Duplicate File Handle
46H	Force Duplicate File Handle
47H	Get Current Directory
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Set Memory Block Size
4BH	Execute Program
4CH	End Process
4DH	Get Return Code of Process
58H	Get or Set Allocation Strategy
62H	Get PSP Address

Redirected INT 21H Functions

Some INT 21H functions are redirected through the INT 2FH interface to external file systems, such as MSCDEX (or the memory cartridge on VIS players), instead of going to the internal FAT file system. These functions assume all devices in the system are redirected. The following table lists the redirected INT 21H functions:

Register AH	Function
39H	Create Directory
3AH	Delete Directory
3BH	Set Current Directory
3CH	Create File
3DH	Open File
3EH	Close File
3FH	Read File or Device
40H	Write File or Device
41H	Delete File
42H	Set File Pointer
43H	Get or Set File Attributes
4EH	Find First File
4FH	Find Next File

MCI Error Codes

The following table describes each of the MCI error return codes and gives the corresponding MMSYSTEM.H constant:

Error	Corresponding Constant and Error Description
0x0101	MCIERR_INVALID_DEVICE_ID Invalid device ID.
0x0103	MCIERR_UNRECOGNIZED_KEYWORD Unknown keyword.
0x0105	MCIERR_UNRECOGNIZED_COMMAND Unknown command.
0x0106	MCIERR_HARDWARE Hardware error on media device.
0x0107	MCIERR_INVALID_DEVICE_NAME The device is not open or is not known.
0x0108	MCIERROR_OUT_OF_MEMORY Not enough memory for requested operation.
0x0109	MCIERR_DEVICE_OPEN The device is already open and is not shareable.
0x010A	MCIERR_CANNOT_LOAD_DRIVER The requested driver is not specified in the SYSTEM.INI file.
0x010B	MCIERR_MISSING_COMMAND_STRING A string command was found without the required parameter.
0x010C	MCIERR_PARAM_OVERFLOW The output string is not long enough.
0x010C	MCIERR_MISSING_STRING_ARGUMENT A required string argument is not present.
0x010D	MCIERR_BAD_INTEGER A non-integral value was encountered when an integer was required.
0x010E	MCIERROR_PARSER_INTERNAL Internal parser error.
0x0110	MCIERR_DRIVER_INTERNAL Internal driver error.
0x0111	MCIERR_MISSING_PARAMETER A required parameter is not present.
0x0112	MCIERR_UNSUPPORTED_FUNCTION Action not available for this device.

Continued

Error	Corresponding Constant and Error Description
0x0113	MCIERR_FILE_NOT_FOUND Requested file not found.
0x0114	MCIERR_DEVICE_NOT_READY Device not ready.
0x0115	MCIERR_INTERNAL Internal error.
0x0116	MCIERR_DRIVER Unspecified device error.
0x0117	MCIERR_CANNOT_USE_ALL The device name "all" is not allowed for this command.
0x0118	MCIERR_MULTIPLE Errors occurred in more than one device.
0x0119	MCIERR_EXTENSION_NOT_FOUND The specified extension was not found.
0x011A	MCIERR_OUTOFRANGE Parameter value out of range.
0x011C	MCIERR_FLAGS_NOT_COMPATIBLE Incompatible parameters were specified.
0x011E	MCIERR_FILE_NOT_SAVED The file was not saved.
0x011F	MCIERR_DEVICE_TYPE_REQUIRED The device name must be a valid device type.
0x0120	MCIERR_DEVICE_LOCKED The device is locked until it is closed automatically.
0x0121	MCIERR_DUPLICATE_ALIAS The specified alias is already in use.
0x0122	MCIERR_BAD_CONSTANT A variable was encountered where a constant was required.
0x0123	MCIERR_MUST_USE_SHAREABLE The specified device was not opened as a shareable device.
0x0124	MCIERR_MISSING_DEVICE_NAME Invalid device name, alias, or filename.
0x0125	MCIERR_BAD_TIME_FORMAT Illegal value for time format.

Continued

Error	Corresponding Constant and Error Description
0x0126	MCIERR_NO_CLOSING_QUOTE No closing quote found for a previous opening quote.
0x0127	MCIERR_DUPLICATE_FLAGS Specified flag already in use on a previous command.
0x0128	MCIERR_INVALID_FILE Invalid file format.
0x0129	MCIERR_NULL_PARAMETER_BLOCK Parameter block pointer was NULL.
0x012A	MCIERR_UNNAMED_RESOURCE Attempt to save unnamed file.
0x012B	MCIERR_NEW_REQUIRES_ALIAS Attempt to create a new object without specifying either a filename or an alias.
0x012C	MCIERR_NOTIFY_ON_AUTO_OPEN The notify flag is illegal on auto open.
0x012D	MCIERR_NO_ELEMENT_ALLOWED Cannot specify a filename with this type of device.
0x012E	MCIERR_NONAPPLICABLE_FUNCTION Invalid command sequence.
0x012F	MCIERR_ILLEGAL_FOR_AUTO_OPEN Generated an illegal message for an auto opened device.
0x0130	MCIERR_FILENAME_REQUIRED The specified filename is not valid.
0x0131	MCIERR_EXTRA_CHARACTERS Extraneous characters found following a closing quote.
0x0132	MCIERR_DEVICE_NOT_INSTALLED The driver for the requested device is not installed.
0x0133	MCIERR_GET_CD The requested directory name does not exist.
0x0134	MCIERR_SET_CD Attempted to select a directory name that does not exist.
0x0135	MCIERR_SET_DRIVE Attempted to change to a drive name that does not exist.
0x0136	MCIERR_DEVICE_LENGTH Driver or device name is too long.

Continued

Error	Corresponding Constant and Error Description
0x0137	MCIERR_DEVICE_ORD_LENGTH Driver or device name is too long.
0x0138	MCIERR_NO_INTEGER Non-integer parameter specified where an integer is required.
0x015A	MCIERR_NO_WINDOW There is no display window.
0x015B	MCIERR_CREATEWINDOW Could not create or use window.
0x015C	MCIERR_FILE_READ A read from the file failed.
0x015D	MCIERR_FILE_WRITE A write to the file failed.
MIDI Sequencer Device Errors	
0x0150	MCIERR_SEQ_DIV_INCOMPATIBLE Set Song Pointer incompatible with SMPTE files.
0x0151	MCIERR_SEQ_PORT_INUSE Specified port is in use.
0x0152	MCIERR_SEQ_PORT_NONEXISTENT Specified port does not exist.
0x0153	MCIERR_SEQ_PORT_MAPNODEVICE Current map uses non-existent device.
0x0154	MCIERR_SEQ_PORT_MISCERROR Miscellaneous error with specified port.
0x0155	MCIERR_SEQ_TIMER Timer error.
0x0156	MCIERR_SEQ_PORTUNSPECIFIED No current MIDI port.
0x0157	MCIERROR_NOMIDIPRESENT No MIDI device found.
Waveform Audio Device Errors	
0x0140	MCIERR_WAVE_OUTPUTSINUSE No compatible waveform-playback device is free.
0x0141	MCIERR_WAVE_SETOUTPUTINUSE Set waveform-playback device is in use.
0x0142	MCIERR_WAVE_INPUTSINUSE No compatible waveform-recording device is free.

Continued

Error	Corresponding Constant and Error Description
0x0143	MCIERR_WAVE_SETINPUTINUSE Set waveform-recording device is in use.
0x0144	MCIERR_WAVE_OUTPUTUNSPECIFIED Any compatible waveform-playback device can be used.
0x0145	MCIERR_WAVE_INPUTUNSPECIFIED Any compatible waveform-recording device can be used.
0x0146	MCIERR_WAVE_OUTPUTSUNSUITABLE No compatible waveform-playback devices.
0x0147	MCIERR_WAVE_SETOUTPUTUNSUITABLE Set waveform-playback device is incompatible with set format.
0x0148	MCIERR_WAVE_INPUTSUNSUITABLE No compatible waveform-recording devices.
0x0149	MCIERR_WAVE_SETINPUTUNSUITABLE Set waveform-recording device is incompatible with set format.

CHAPTER 5

Function Directory

This chapter contains an alphabetical listing of the new Modular Windows functions and macros. Each entry contains the following items:

- The type, name, and description of input parameters
- The syntax for the function or macro
- The purpose of the function or macro
- A description of the return value
- Optional comments about using the function or macro
- Optional examples showing how to use the function or macro
- Optional cross references to other functions, macros, messages, and data structures

DisplayDib

Syntax

WORD DisplayDib(*lpbi*, *lpBits*, *wFlags*)

The **DisplayDib** function displays a bitmap using the full screen, centering the bitmap and clipping it if necessary. Normally, control does not return to the application until the user initiates an action such as pressing a hand-control button, keyboard key, or mouse button.

To call **DisplayDib**, an application must be the active application. All inactive applications, GDI screen updates, and MCI streaming audio services (waveform and MIDI sequencer) are suspended while **DisplayDib** has control of the display.

Parameters

LPBITMAPINFO *lpbi*

Specifies a pointer to a **BITMAPINFO** structure describing the bitmap to be displayed.

LPSTR *lpBits*

Specifies a pointer to the bitmap bits. If this parameter is NULL, the bits are assumed to follow the **BITMAPINFO** structure pointed to by *lpbi*.

WORD *wFlags*

Specifies options for displaying the bitmap using a combination of the following flags:

DISPLAYDIB_NOCENTER

Display the image without centering it. The image appears in the lower-left corner of the screen.

DISPLAYDIB_NOWAIT

Return immediately after displaying the bitmap.

DISPLAYDIB_BEGIN

Prepare to display a series of bitmaps using the same video mode with multiple calls to **DisplayDib**. Subsequent calls to **DisplayDib** will not set the video mode, eliminating the screen flicker that occurs with video-mode changes. If the DISPLAYDIB_MODE_DEFAULT flag is specified, **DisplayDib** chooses the video mode based on the **BITMAPINFO** structure specified by the *lpbi* parameter. The *lpBits* parameter is ignored.

Video memory is not cleared for subsequent **DisplayDib** calls. If the new image does not completely cover the previous image, remnants of the previous image will remain on the screen.

DISPLAYDIB_END

Indicates the end of multiple calls to **DisplayDib**. The *lpbi* and *lpBits* parameters should be set to NULL when specifying this flag.

DISPLAYDIB_NOPALETTE

Displays the image without changing the palette. Use this flag to display a series of DIBs that use a common palette.

DISPLAYDIB_ZOOM2
Stretches the bitmap by a factor of two before displaying it. This option works only with 8-bit palettized DIBs (DISPLAYDIB_MODE_320x200x8). RLE bitmaps should not be stretched unless the entire stretched image will fit on screen.

DISPLAYDIB_TEST
Query to determine if **DisplayDib** supports the given video mode—does not set the video mode or the palette and does not wait before returning.

DISPLAYDIB_MODE_DEFAULT
Display the bitmap using the video mode appropriate for the given bitmap. The video mode is determined from the **BITMAPINFO** structure specified by the *lpbi* parameter.

DISPLAYDIB_MODE_320x200x8
Display an 8-bit palettized DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_555_320x200
Display an RGB555 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_555_320x400
Display an RGB555 DIB using 320-by-400 resolution.

DISPLAYDIB_MODE_565_320x200
Display an RGB565 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_565_320x400
Display an RGB565 DIB using 320-by-400 resolution.

DISPLAYDIB_MODE_YUV16_320x200
Display a YUV16 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_YUV16_320x400
Display a YUV16 DIB using 320-by-400 resolution.

DISPLAYDIB_MODE_YUV8_320x200
Display a YUV8 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_YUV8_320x400
Display a YUV8 DIB using 320-by-400 resolution.

Return Value

The return value is zero if successful; otherwise, it is one of the following error codes:

DISPLAYDIB_NOTSUPPORTED
DisplayDib does not support the specified video mode.

DISPLAYDIB_INVALIDDIB
The bitmap specified by *lpbi* is not a valid bitmap.

DISPLAYDIB_INVALIDFORMAT
The bitmap specified by *lpbi* specifies a type of bitmap that is not supported.

DISPLAYDIB_INVALIDTASK
The caller is an inactive application. **DisplayDib** can only be called by an active application.

DISPLAYDIB_FEATURE_NOTSUPPORTED
The specified option is not supported. Currently, this error is returned only when the DISPLAYDIB_ZOOM2 option is specified for a bitmap that is not an 8-bit palettized DIB.

DISPLAYDIB_DIBMODEMISMATCH
The given DIB cannot be shown in the specified video mode.

DISPLAYDIB_WIDEDIB
The given DIB is too wide to display. This includes the cases when a YUV8 DIB is to be displayed off the left edge of the screen and when a RLE8 DIB is wider than 320 (or 160 when DISPLAYDIB_ZOOM2 is also specified).

Comments

The **DisplayDib** function displays bitmaps described with the Windows **BITMAPINFO** data structure in either BI_RGB or BI_RLE8 format; it does not support bitmaps described with the OS/2 **BITMAPCOREHEADER** data structure.

Because **DisplayDib** changes the video mode and disables the display driver, applications cannot call GDI functions while **DisplayDib** has control of the display.

Due to the differential encoding used in the YUV8 format, YUV8 DIBs must be displayed such that the left edge of the DIB is aligned with the left edge of the screen, otherwise the image will not be displayed correctly.

For better performance, RLE8 DIBs are clipped only in the y-direction.

See Also

DisplayDibEx, EnterDVA

DisplayDibEx

Syntax

WORD DisplayDibEx(*lpbi*, *lpBits*, *x*, *y*, *wFlags*)

The **DisplayDibEx** function displays a bitmap using the full screen, centering the bitmap and clipping it, if necessary, or placing it at a specified position on the screen. Normally, control does not return to the application until the user initiates an action such as pressing a hand-control button, keyboard key, or mouse button.

To call **DisplayDibEx**, an application must be the active application. All inactive applications, GDI screen updates, and MCI streaming audio services (waveform and MIDI sequencer) are suspended while **DisplayDibEx** has control of the display.

Parameters

LPBITMAPINFO *lpbi*
Specifies a pointer to a **BITMAPINFO** structure describing the bitmap to be displayed.

LPSTR *lpBits*
Specifies a pointer to the bitmap bits. If this parameter is NULL, the bits are assumed to follow the **BITMAPINFO** structure pointed to by *lpbi*.

int *x*
Specifies the *x*-coordinate of the screen location to place the origin of the bitmap. The origin is the lower-left corner of the bitmap.

int *y*
Specifies the *y*-coordinate of the screen location to place the origin of the bitmap. The origin is the lower-left corner of the bitmap.

WORD *wFlags*
Specifies options for displaying the bitmap using a combination of the following flags:

DISPLAYDIB_NOWAIT
Return immediately after displaying the bitmap.

DISPLAYDIB_BEGIN
Prepare to display a series of bitmaps using the same video mode with multiple calls to **DisplayDibEx**. Subsequent calls to **DisplayDibEx** will not set the video mode, eliminating the screen flicker that occurs with video mode changes. If the DISPLAYDIB_MODE_DEFAULT flag is specified, **DisplayDibEx** chooses the video mode based on the **BITMAPINFO** structure specified by the *lpbi* parameter. The *lpBits* parameter is ignored.

Video memory is not cleared for subsequent **DisplayDibEx** calls. If the new image does not completely cover the previous image, remnants of the previous image will remain on the screen.

DISPLAYDIB_END
Indicates the end of multiple calls to **DisplayDibEx**. The *lpbi* and *lpBits* parameters should be set to NULL when specifying this flag.

DISPLAYDIB_NOPALETTE
Displays the image without changing the palette. Use this flag to display a series of DIBs that use a common palette.

DISPLAYDIB_ZOOM2
Stretch the bitmap by a factor of two before displaying it. This option works only with 8-bit palettized DIBs (DISPLAYDIB_MODE_320x200x8). RLE bitmaps should not be stretched unless the entire stretched image will fit on the screen.

DISPLAYDIB_TEST
Query to determine if **DisplayDibEx** supports the given video mode—does not set the video mode or the palette and does not wait before returning.

DISPLAYDIB_MODE_DEFAULT
Display the bitmap using the video mode appropriate for the given bitmap. The video mode is determined from the **BITMAPINFO** structure specified by the *lpbi* parameter.

DISPLAYDIB_MODE_320x200x8
Display an 8-bit palettized DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_555_320x200
Display an RGB555 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_555_320x400
Display an RGB555 DIB using 320-by-400 resolution.

DISPLAYDIB_MODE_565_320x200
Display an RGB565 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_565_320x400
Display an RGB565 DIB using 320-by-400 resolution.

DISPLAYDIB_MODE_YUV16_320x200
Display a YUV16 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_YUV16_320x400
Display a YUV16 DIB using 320-by-400 resolution.

DISPLAYDIB_MODE_YUV8_320x200
Display a YUV8 DIB using 320-by-200 resolution.

DISPLAYDIB_MODE_YUV8_320x400
Display a YUV8 DIB using 320-by-400 resolution.

Return Value

The return value is zero if successful; otherwise, it is one of the following error codes:

DISPLAYDIB_NOTSUPPORTED
DisplayDibEx does not support the specified video mode.

DISPLAYDIB_INVALIDDIB
The bitmap specified by *lpbi* is not a valid bitmap.

DISPLAYDIB_INVALIDFORMAT
The bitmap specified by *lpbi* specifies a type of bitmap that is not supported.

DISPLAYDIB_INVALIDTASK
The caller is an inactive application. **DisplayDibEx** can only be called by an active application.

DISPLAYDIB_FEATURE_NOTSUPPORTED
The specified option is not supported. Currently, this error is returned only when the DISPLAYDIB_ZOOM2 option is specified for a bitmap that is not an 8-bit palettized DIB.

DISPLAYDIB_DIBMODEMISMATCH
The given DIB cannot be shown in the specified video mode.

DISPLAYDIB_WIDEDIB
The given DIB is too wide to display. This includes the cases when a YUV8 DIB is to be displayed off the left edge of the screen and when a RLE8 DIB is wider than 320 pixels (160 pixels when DISPLAYDIB_ZOOM2 is also specified).

Comments

The **DisplayDibEx** function displays bitmaps described by the Windows **BITMAPINFO** data structure in either BI_RGB or BI_RLE8 format; it does not support bitmaps described with the OS/2 **BITMAPCOREHEADER** data structure.

Because **DisplayDibEx** changes the video mode and disables the display driver, applications cannot call GDI functions while **DisplayDibEx** has control of the display.

With YUV16 DIBs, the *x*-coordinate specified in the *x* parameter will be rounded to the nearest multiple of 2. With YUV8 DIBs, the *x*-coordinate will be rounded to the nearest multiple of 4. Also, when the DISPLAYDIB_ZOOM2 flag is specified, **DisplayDibEx** will truncate the *x*-coordinate of the origin to be even-aligned.

Due to the differential encoding used in the YUV8 format, YUV8 DIBs must be displayed such that the left edge of the DIB is aligned with the left edge of the screen, otherwise the image will not be displayed correctly.

For better performance, RLE8 DIBs are clipped only in the y-direction.

See Also

DisplayDib, **EnterDVA**

EnterDVA

Syntax

WORD EnterDVA(*wFlags*)

The **EnterDVA** macro begins direct video access (DVA) by suspending GDI and the currently running display driver and setting the video hardware to the specified video mode.

Parameters

WORD *wFlags*
Specifies options using a combination of the following flags:

DISPLAYDIB_NOWAIT
Return immediately after beginning direct video access.

DVA_MODE_320x200x8
Sets the video mode to TVVGA, 320-by-200 resolution.

DVA_MODE_555_320x200
Sets the video mode to RGB555, 320-by-200 resolution.

DVA_MODE_555_320x400
Sets the video mode to RGB555, 320-by-400 resolution.

DVA_MODE_565_320x200
Sets the video mode to RGB565, 320-by-200 resolution.

DVA_MODE_565_320x400
Sets the video mode to RGB565, 320-by-400 resolution.

DVA_MODE_YUV16_320x200
Sets the video mode to YUV16, 320-by-200 resolution.

DVA_MODE_YUV16_320x400
Sets the video mode to YUV16, 320-by-400 resolution.

DVA_MODE_YUV8_320x200
Sets the video mode to YUV8, 320-by-200 resolution.

DVA_MODE_YUV8_320x400
Sets the video mode to YUV8, 320-by-400 resolution.

Return Value

The return value is zero if the function is successful; otherwise, it is one of the following error codes:

DISPLAYDIB_NOTSUPPORTED
EnterDVA does not support the specified video mode.

DISPLAYDIB_INVALIDTASK
The caller is an inactive application. **EnterDVA** can only be called by an active application.

Comments While in DVA mode, applications can call **DisplayDib** using the DISPLAYDIB_CONTINUE flag to display DIBs without changing the video mode.

Example The following code fragment shows how to get pointers to video memory:

```
// Pointers to base of video memory and base of video-overlay memory
LPVOID lpA000;    // far pointer to base of video memory
LPVOID lpB000;    // far pointer to base of video overlay

// External variables defined in the Windows Kernel module
extern WORD _A000h;  // selector for video memory
extern WORD _B000h;  // selector for video overlay

// Convert selectors into far pointers
lpA000 = (LPVOID) MAKELONG(0L, &_A000h);
lpB000 = (LPVOID) MAKELONG(0L, &_B000h);
```

See Also **LeaveDVA, DisplayDib, DisplayDibEx**

fmAddWindow

Syntax **int fmAddWindow**(*hWnd*)

The **fmAddWindow** function adds a control to the list of controls managed by the focus manager.

Parameters

HWND *hWnd*
Specifies the window handle of the control to add.

Return Value If the function is successful, the return value is the number of windows currently being managed by the focus manager; otherwise, the return value is the following error code:

FM_ERR
The specified window is already in the focus-manager list or the focus-manager list is full (maximum of 255 entries).

Comments Controls based on the predefined control classes (TVBUTTON, TVLISTBOX, TVSCROLLBAR, TVSCROLLPAD, TVSPINBUTTON, TVKEYBOARD, and TVEDITBOX) are automatically added to the focus manager when they are created. The **fmAddWindow** function should only be used with custom controls.

Controls added to the focus manager must have the WS_VISIBLE and WS_TABSTOP window style bits set and the WS_DISABLE bit cleared to get the focus. Controls without these bits set will be ignored when the focus manager changes the focus.

See Also **fmDeleteWindow**

fmDeleteWindow

Syntax **BOOL fmDeleteWindow**(*hWnd*)

The **fmDeleteWindow** function removes a control from the list of controls managed by the focus manager.

Parameters

HWND *hWnd*
Specifies the window handle of the control to remove.

Return Value If the function is successful, the return value is the number of windows currently being managed by the focus manager; otherwise, the return value is the following error code:

FM_ERR
The specified window is not in the focus-manager list.

Comments Applications can use **fmDeleteWindow** to remove controls based on the predefined control classes (TVBUTTON, TVLISTBOX, TVSCROLLBAR, TVSCROLLPAD, TVSPINBUTTON, TVKEYBOARD, and TVEDITBOX) from the list of controls managed by the focus manager.

To temporarily prevent a control from getting the focus, applications can clear the WS_TABSTOP window style bit by calling the **SetWindowLong** function with the *nOffset* parameter set to GWL_STYLE.

See Also **fmAddWindow**

fmGetLastCursorPos

Syntax **void fmGetLastCursorPos**(*lppt*)

The **fmGetLastCursorPos** function retrieves the last position of the cursor prior to the focus change. This function relies on **fmSetCursorPos** being called to set the current position.

Parameters

LPPOINT *lppt*
Specifies a pointer to a **POINT** structure that will be filled with the last cursor position.

See Also **fmSetCursorPos**

fmGetLastDirection

Syntax **WORD fmGetLastDirection**()

The **fmGetLastDirection** function retrieves the direction of the last focus movement. Custom controls with multiple focus areas can use this function to determine which area to initially give the focus.

Return Value The return value is the direction of the last focus movement specified by one of the hand-control virtual-key codes.

fmGetWindowVectors

Syntax **BOOL fmGetWindowVectors**(*hWnd*, *lpDV*)

The **fmGetWindowVectors** function retrieves the focus vectors for a control managed by the focus manager. Focus vectors determine which control gets the focus when the focus is moved from the given control.

Parameters

HWND *hWnd*
Specifies the window handle of the control for which to retrieve the focus vectors.

LPDIRVECTORS *lpDV*
Specifies a far pointer to a **DIRVECTORS** structure that will be filled with the focus vectors.

Return Value The return value is nonzero if the function is successful; otherwise, it is zero.

See Also **fmSetWindowVectors**, **DIRVECTORS**

fmIsFocusMessage

Syntax

BOOL fmIsFocusMessage(*hWnd*, *uiMsg*, *wParam*, *lParam*)

The **fmIsFocusMessage** function sends WM_KEYDOWN messages to the focus manager for processing.

Parameters

HWND *hWnd*
Specifies the window handle of the control that received the WM_KEYDOWN message.

UINT *uiMsg*
Specifies the message. This parameter should be set to WM_KEYDOWN.

WPARAM *wParam*
Specifies the WORD parameter associated with the WM_KEYDOWN message.

LPARAM *lParam*
Specifies the LONG parameter associated with the WM_KEYDOWN message.

Return Value

The return value is TRUE if the focus manager processes the WM_KEYDOWN message and sets the focus to a new window; otherwise, it is FALSE.

Comments

This function should only be used in the window procedures of custom controls that have been added to the focus manager using **fmAddWindow**. Custom controls can process WM_KEYDOWN messages without passing them to the focus manager. For example, a control such as the TV list box can have several areas within the control that can each have the focus. In this case, the control only calls **fmIsFocusMessage** when it determines that the focus is being moved to another control.

See Also

fmGetLastDirection

fmSetCursorPos

Syntax **void fmSetCursorPos**(*lppt*)

The **fmSetCursorPos** function sets the position of the cursor on the screen to indicate which window has the current keyboard or hand-control focus.

Parameters

LPPOINT *lppt*
Specifies a pointer to a **POINT** structure containing the new cursor position.

See Also **fmGetLastCursorPos**

fmSetWindowVectors

Syntax **BOOL fmSetWindowVectors**(*hWnd*, *lpDV*)

The **fmSetWindowVectors** function sets the focus vectors for a control managed by the focus manager. Focus vectors determine which control gets the focus when the focus is moved from the given control.

Parameters

HWND *hWnd*
Specifies the window handle of the control for which to set the focus vectors.

LPDIRVECTORS *lpDV*
Specifies a far pointer to a **DIRVECTORS** structure containing the focus vectors. The focus manager makes a copy of this structure—applications can free this memory after calling **fmSetWindowVectors**.

Return Value The return value is nonzero if the function is successful; otherwise, it is zero.

Comments The **DIRVECTORS** structure uses window handles to identify controls. If an entry contains NULL, the focus manager does not tab to any other control when the hand-control direction key corresponding to the entry is pressed. If the entry contains FM_DEFAULT, the focus is changed to the next window based on the default algorithm. This algorithm uses the screen rectangles of siblings of the window that currently has the focus to determine the appropriate window to receive the focus.

See Also **fmGetWindowVectors**, **DIRVECTORS**

fmTranslateHCKey

Syntax

BOOL fmTranslateHCKey(*lpw*, *l*)

The **fmTranslateHCKey** function selectively translates WM_KEYDOWN and WM_KEYUP messages that are generated by the hand control into raw, unmapped virtual-key codes.

Parameters

LPWORD *lpw*

Specifies the *wParam* parameter of the key message. This is the parameter that can be changed by **fmTranslateHCKey**.

LONG *l*

Specifies the *lParam* parameter of the key message. This parameter contains the scan-code information used by **fmTranslateHCKey** to determine whether the key message was generated by the hand control or an actual keyboard.

Return Value

The return value is TRUE if the scan code came from a hand control and the virtual-key code was changed; otherwise, the return value is FALSE.

Comments

This function should only be used in the window procedure of custom controls that have been added to the focus manager using **fmAddWindow**. It should be called immediately before WM_KEYDOWN messages are processed and passed to **fmIsFocusMessage**. The **fmTranslateHCKey** function translates hand-control key messages into raw virtual-key codes to make them appear as if they came from the keyboard. This is necessary because the hand-control might map the various hand-control keys to arbitrary virtual-key codes. Child controls use this function as a method for interpreting raw unmapped hand-control codes. If a scan code is found that is not generated by the hand control, or if it is from a hand-control button that is not interpreted by the focus manager (buttons other than direction or action buttons), or if random roam is enabled for the player that generated the event, then the *wParam* parameter will not be changed.

See Also

fmIsFocusMessage

HC_IS_HC

Syntax

BOOL HC_IS_HC()

The **HC_IS_HC** macro determines if the current key event was generated by the hand control.

Return Value

The return value is TRUE if the given key event is from the hand control; otherwise, it is FALSE.

Comments This macro is only valid when called during processing of WM_KEYUP or WM_KEYDOWN messages. WM_KEYDOWN messages always precede WM_CHAR messages. To determine if a WM_CHAR event was generated by the hand control, call **HC_IS_HC** when the WM_KEYDOWN message is received and save the result to use with the next WM_CHAR message.

See Also **HC_PLAYER**

HC_KEY_OFFSET

Syntax **WORD HC_KEY_OFFSET**(*wVkey*)

The **HC_KEY_OFFSET** macro returns the offset for a given hand-control button. Use this offset to specify hand-control buttons when remapping buttons using the **hcControl** function.

Parameters WORD *wVkey*
Specifies the unmapped virtual-key code identifying the hand-control button.

Return Value The return value is the offset corresponding to the given hand-control button.

Comments Hand-control key offsets are identical to the scan codes associated with WM_KEYUP and WM_KEYDOWN messages. Scan codes can be examined when an application needs to know what physical button was pressed, regardless of the key mapping.

See Also **hcControl**

HC_PLAYER

Syntax **BOOL HC_PLAYER**(*wVkey*, *bScan*)

The **HC_PLAYER** macro determines if a given hand-control key event is from the first player or the second player.

Parameters WORD *wVkey*
Specifies the virtual-key code of the key event.

BYTE *bScan*
Specifies the scan code of the key event.

Return Value The return value is zero if the key event is from the first player; one if the key event is from the second player.

Comments Use this macro only with hand-control key events.

See Also **HC_IS_HC**

HC_VKN2VK

Syntax **WORD HC_VKN2VK**(*wVkey*, *bScan*)

The **HC_VKN2VK** macro returns a virtual-key code for the first player given a hand-control key event for the first or the second player.

Parameters

WORD *wVkey*
Specifies the virtual-key code of the key event.

BYTE *bScan*
Specifies the scan code of the key event.

Return Value The return value is a virtual-key code for the first player.

Comments Use this macro only with hand-control key events.

See Also **HC_IS_HC**

hcControl

Syntax **DWORD hcControl**(*wFlag*, *wParam*, *lParam*)

The **hcControl** function changes settings for the hand-control driver.

Parameters

WORD *wFlag*
Specifies the setting to change using one of the following flags:

HC_SET_ROAM
Enables and disables roaming mode. Set *wParam* to TRUE to enable roaming mode, or FALSE to disable roaming mode. The default state is disabled.

To enable roaming mode for both hand controls, set *lParam* to zero. To enable roaming mode for individual players, set *lParam* to HC_ROAM1 for player 1 or HC_ROAM2 for player 2. If *wParam* is FALSE, roaming mode is disabled for both players—the *lParam* parameter is ignored.

HC_GET_ROAM
Retrieves the random-roam setting. Use the HC_ROAM1 and HC_ROAM2 constants to test the bits corresponding to each hand control.

HC_SET_INIT
Sets the initial velocity of the cursor when roaming mode is enabled. Specify the initial velocity in the *wParam* parameter. The default initial velocity is 1. The maximum initial velocity is 50.

HC_GET_INIT
Retrieves the initial cursor-velocity setting. The setting is returned in the low-order word of the return value.

HC_SET_MAX
Sets the maximum velocity of the cursor when roaming mode is enabled. Specify the maximum velocity in the *wParam* parameter. The default maximum velocity is 50. The maximum value is 50.

HC_GET_MAX
Retrieves the maximum cursor velocity setting. The setting is returned in the low-order word of the return value.

HC_SET_RATE
Sets the cursor-update rate when roaming mode is enabled. On each cursor update, a new position is calculated from the current cursor velocity and acceleration. Use the *wParam* parameter to specify the update rate. The default rate is 20 updates per second. The maximum rate is 30 updates per second.

HC_GET_RATE
Retrieves the cursor-update rate setting. The setting is returned in the low-order word of the return value.

HC_SET_ACCEL
Sets the cursor acceleration when roaming mode is enabled. This value is added to the cursor velocity on each update of the cursor position. Specify the acceleration in the *wParam* parameter. The default acceleration is 1. The maximum acceleration is 12.

HC_GET_ACCEL
Retrieves the cursor-acceleration setting. The setting is returned in the low-order word of the return value.

HC_SET_KEYMAP
Sets the key map for the hand-control buttons. Use the *lParam* parameter to specify a far pointer to an array of 20 bytes containing the new key map.

HC_GET_KEYMAP
Retrieves the current key map for the hand-control buttons. Use the *lParam* parameter to specify a far pointer to an array of 20 bytes that will be filled with the virtual-key codes for the current key map.

HC_SET_KEY
Sets a single entry in the key map for the hand-control buttons. Use the *wParam* parameter to specify the offset for the entry you want to change and the *lParam* parameter to specify the new virtual-key code.

HC_GET_KEY
Gets a single entry in the key map for the hand-control buttons. Specify the offset for the entry to change in the *wParam* parameter. The low-order word of the return value contains the virtual-key code for the given entry.

WPARAM *wParam*
Specifies a value dependent on the flag passed in the *wFlags* parameter.

LPARAM *lParam*
Specifies a value dependent on the flag passed in the *wFlags* parameter.

Return Value

The return value is dependent on the flag specified in the *wParam* parameter. If one of the parameters is invalid, the return value is HCERR_BADPARAMETER.

Comments

Use the **HC_KEY_OFFSET** macro to get key offset values to use when setting and getting key-map entries. Out-of-range values are converted to a minimum or maximum allowable value.

Example

The following code fragment uses the **hcControl** function and the **HC_KEY_OFFSET** macro to set a new key map for the hand-control buttons:

```
BYTE acKeymap[20];

// Set up player 1 key map
acKeymap[ HC_KEY_OFFSET( VK_HC1_PRIMARY )] = VK_LBUTTON;
acKeymap[ HC_KEY_OFFSET( VK_HC1_SECONDARY)] = VK_RBUTTON;
acKeymap[ HC_KEY_OFFSET( VK_HC1_UP    )] = VK_UP;
acKeymap[ HC_KEY_OFFSET( VK_HC1_LEFT  )] = VK_LEFT;
acKeymap[ HC_KEY_OFFSET( VK_HC1_RIGHT  )] = VK_RIGHT;
acKeymap[ HC_KEY_OFFSET( VK_HC1_DOWN  )] = VK_DOWN;
acKeymap[ HC_KEY_OFFSET( VK_HC1_F1    )] = VK_RETURN;
acKeymap[ HC_KEY_OFFSET( VK_HC1_F2    )] = VK_TAB;
acKeymap[ HC_KEY_OFFSET( VK_HC1_TOOLBAR )] = VK_MENU;
acKeymap[ HC_KEY_OFFSET( VK_HC1_F4    )] = VK_ALPHANUM('Z');

// Set up player 2 key map
acKeymap[ HC_KEY_OFFSET( VK_HC2_PRIMARY )] = VK_SPACE;
acKeymap[ HC_KEY_OFFSET( VK_HC2_SECONDARY)] = VK_ESCAPE;
acKeymap[ HC_KEY_OFFSET( VK_HC2_UP    )] = VK_ALPHANUM('A');
acKeymap[ HC_KEY_OFFSET( VK_HC2_LEFT  )] = VK_ALPHANUM('B');
acKeymap[ HC_KEY_OFFSET( VK_HC2_RIGHT  )] = VK_ALPHANUM('C');
acKeymap[ HC_KEY_OFFSET( VK_HC2_DOWN  )] = VK_ALPHANUM('D');
acKeymap[ HC_KEY_OFFSET( VK_HC2_F1    )] = VK_ALPHANUM('E');
acKeymap[ HC_KEY_OFFSET( VK_HC2_F2    )] = VK_ALPHANUM('F');
acKeymap[ HC_KEY_OFFSET( VK_HC2_F3    )] = VK_ALPHANUM('G');
acKeymap[ HC_KEY_OFFSET( VK_HC2_F4    )] = VK_ALPHANUM('H');

// Set the new key map
hcControl( HC_SET_KEYMAP, 0, (DWORD)(LPSTR)acKeymap);
```

See Also **HC_KEY_OFFSET**

hcGetCursorPos

Syntax **void hcGetCursorPos**(*lpp*)

The **hcGetCursorPos** function returns the current cursor position.

Parameters LPPOINT *lpp*
Specifies a pointer to a **POINT** structure that will be filled with the cursor location.

Comments Applications written for the hand control should use this function instead of the **GetCursorPos** function to ensure future compatibility.

See Also **hcControl, hcSetCursorPos**

hcSetCursorPos

Syntax **void hcSetCursorPos**(*x*, *y*, *bSmooth*)

The **hcSetCursorPos** function moves the cursor to a given position. The cursor can be moved instantly to the new position or moved smoothly from the current position to the new position. Moving the cursor smoothly allows users to follow a change of focus from one control to another.

Parameters int *x*
Specifies the new x-position of the cursor, in screen coordinates.

int *y*
Specifies the new y-position of the cursor, in screen coordinates.

BOOL *bSmooth*
Specifies whether to move the cursor instantly or smoothly from the current position. Set *bSmooth* to TRUE to move the cursor smoothly; FALSE to move the cursor instantly.

Comments Applications written for the hand control should use this function instead of the **SetCursorPos** function to ensure future compatibility.

See Also **hcControl, hcGetCursorPos**

LeaveDVA

Syntax

WORD LeaveDVA(*wFlags*)

The **LeaveDVA** macro ends direct video access and enables GDI and the display driver. The current task is unlocked and all windows are invalidated, causing them to be repainted. Normally, control does not return to the application until the user initiates an action such as pressing a hand-control button, keyboard key, or mouse button.

Parameters

WORD *wFlags*
Specifies options using the following flag:

DISPLAYDIB_NOWAIT
Return immediately after ending direct video access.

Return Value

The return value is zero if the function is successful; otherwise, it is the following error code:

DISPLAYDIB_INVALIDTASK
The caller is an inactive application. The **LeaveDVA** macro can only be called by an active application.

Comments

The **LeaveDVA** macro should only be called to balance a successful call to **EnterDVA**.

See Also

EnterDVA

tvGetHighlightFrame

Syntax

BOOL tvGetHighlightFrame(*lpColor*, *wOnDuration*, *wOffDuration*)

The **tvGetHighlightFrame** function retrieves the current highlight color and blinking rate used for highlighting controls that have the input focus.

Parameters

COLORREF FAR * *lpColor*
Specifies a pointer to a COLORREF to receive the highlight color value.

LPWORD *wOnDuration*
Specifies a pointer to a WORD to receive the highlight-on duration.

LPWORD *wOffDuration*
Specifies a pointer to a WORD to receive the highlight-off duration.

Return Value The return value is nonzero if the blinking highlight is turned on; it is zero if the blinking is turned off.

See Also **tvSetHighlightFrame**

tvGetStockObject

Syntax **HGDIOBJ tvGetStockObject(***nObject***)**

The **tvGetStockObject** function retrieves a handle to one of the predefined stock objects.

Parameters int *nObject*

Specifies the stock object for which to retrieve a handle. This parameter can be one of the following values:

SMALLFONT
: Small system font (14 point sans serif).

BIGFONT
: Large system font (18 point sans serif).

Return Value The return value is the handle of the requested stock object if the function is successful; otherwise, the return value is NULL.

Example The following code fragment sets the font in a TV static control to the large system font:

```
HWND hWndStatic;
HFONT hFont;

hFont = tvGetStockObject(BIGFONT);
SendMessage(hWndStatic, WM_SETFONT, hFont, 0);
```

tvGetUIFlags

Syntax **WORD tvGetUIFlags()**

The **tvGetUIFlags** function allows applications to get the current state of settings used by the TV user-interface controls.

Return Value The return value is a bit field specifying the current settings of the TV user-interface controls. The following constants correspond to bits in the return value:

TVSMOOTHCURSOR—If set, the focus manager smoothly moves the cursor when changing focus between controls. If cleared, the focus manager moves the cursor with one discrete jump to its new location. The default is to smoothly move the cursor.

TVFASTDRAW—If set, all controls will be drawn directly on the screen with multiple blits. If cleared, controls will be drawn on an off-screen compatible DC, then drawn on the screen with a single blit. The first method, while faster, causes visible flickering while the control is being drawn. The latter method minimizes these effects. The default is to draw to an off-screen DC, then blit the off-screen DC to the screen.

TVFMCHECKPARENT—If set, the focus manager changes the focus only to windows having the same parent as the window that currently has the focus. If cleared, every window managed by the focus manager is eligible to receive the focus. The default is to change the focus only to windows having the same parent as the window that currently has the focus.

TVNOMOVECURSOR—If set, the focus manager won't move the mouse cursor to the control that is receiving the focus. The focus ring will still be displayed to indicate focus.

TVFMNOTAB1—If set, the focus manager will not tab to another control in response to a player 1 direction key.

TVFMNOTAB2—If set, the focus manager will not tab to another control in response to a player 2 direction key.

Comments

The focus manager will not tab to a control if random roam has been enabled for the player that generated the event.

See Also

tvSetUIFlags, hcControl

tvSetHighlightFrame

Syntax

BOOL tvSetHighlightFrame(*color*, *wOnDuration*, *wOffDuration*)

The **tvSetHighlightFrame** function sets the highlight color and blinking rate used for highlighting controls that have the input focus.

Parameters

COLORREF *color*
Specifies the highlight color.

WORD *wOnDuration*
Specifies the highlight-on duration in milliseconds.

WORD *wOffDuration*
Specifies the highlight-off duration in milliseconds.

Return Value The return value is nonzero if the function is successful; it is zero when an error is detected.

Comments To change only the blinking rate, set *color* to DEFAULT_COLOR. To change only the color, set both *wOnDuration* and *wOffDuration* to zero.

When setting the blinking rate, *wOnDuration* must be nonzero. To turn off the blinking effect, set *wOffDuration* to zero and *wOnDuration* to a nonzero value. The default setting for the blinking effect is off.

See Also **tvGetHighlightFrame**

tvSetUIFlags

Syntax **void tvSetUIFlags**(*wNewFlags*)

The **tvSetUIFlags** function allows applications to customize the operation of the TV user-interface controls.

Parameters WORD *wNewFlags*
Specifies a bit field containing settings for the TV user-interface controls. The following constants correspond to bits in the *wNewFlags* parameter:

TVSMOOTHCURSOR—If set, the focus manager smoothly moves the cursor when changing focus between controls. If cleared, the focus manager moves the cursor with one discrete jump to its new location. The default is to smoothly move the cursor.

TVFASTDRAW—If set, all controls will be drawn directly on the screen with multiple blits. If cleared, controls will be drawn on an off-screen compatible DC, then drawn on the screen with a single blit. The first method, while faster, causes visible flickering while the control is being drawn. The latter method minimizes these effects. The default is to draw to an off-screen DC, then blit from the off-screen DC to the screen.

TVFMCHECKPARENT—If set, the focus manager changes the focus only to windows having the same parent as the window that currently has the focus. If cleared, every window managed by the focus manager is eligible for receiving the focus. The default is to change the focus only to windows having the same parent as the window that currently has the focus.

TVNOMOVECURSOR—If set, the focus manager will not move the cursor to controls that receive the focus. The control will still be highlighted to indicate focus. The default is to move the cursor.

TVFMNOTAB1—If set, the focus manager will not tab to another control in response to a player 1 direction key.

TVFMNOTAB2—If set, the focus manager will not tab to another control in response to a player 2 direction key.

Comments

Each of these settings can be changed individually by using **tvGetUIFlags** to retrieve the current settings and then changing only the desired settings before calling **tvSetUIFlags**. The focus manager will not tab to a control if random roam has been enabled for the player that generated the event.

See Also

tvGetUIFlags, **hcControl**

CHAPTER 6

Message Directory

This chapter contains an alphabetical listing of the new Modular Windows messages. Each entry contains the following items:

- The name of the message
- The purpose of the message
- The type, name, and description of the parameters for each message
- A description of the return value, if any
- Optional comments about using the message
- Optional examples showing how to use the message
- Optional cross references to other functions, macros, messages, and data structures

KM_CHAR

The KM_CHAR message notifies an application that a key on a soft-keyboard control has been pressed.

Parameters

WPARAM *wParam*
Specifies the virtual-key code of the key.

LPARAM *lParam*
Specifies the repeat count in the low-order word.

Comments

The virtual-key codes specified in *wParam* are identical to the standard Windows virtual-key codes. The message sequence for soft-keyboard controls is similar to the message sequence for IBM®-style keyboards with Windows.

This message will only be posted by soft-keyboard controls that don't have the KS_WITHTEXT style.

See Also

KM_KEYDOWN, KM_KEYUP, WM_CHAR

KM_GETDEFKEY

The KM_GETDEFKEY message gets the current state of the Caps Lock button on a soft-keyboard control or edit control. It also returns which button has the focus.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value

If the request is successful, the return value is the index of the key with the focus combined (logical OR) with either the SETDEFKEY_CAPSLOCK or SETDEFKEY_CAPSUNLOCK constant; otherwise, the return value is zero.

See Also

KM_SETPROMPT, KM_SETDEFKEY

KM_GETPROMPT

The KM_GETPROMPT message is sent by applications to KS_WITHTEXT style soft-keyboard controls or edit boxes to get the text that is displayed in the prompt area.

Parameters

WPARAM *wParam*
Specifies the length of the buffer that receives the prompt text.

LPARAM *lParam*
Specifies a pointer (LPSTR) to a buffer that receives the prompt text as a null-terminated string.

Return Value

The return value is the number of bytes copied, excluding the null terminator, if the request is successful; otherwise, the return value is zero. The return value is zero only when the length of the buffer is smaller than needed.

Comments

To display a prompt, a soft-keyboard control or an edit box must be KS_WITHTEXT style. The prompt of a KS_WITHTEXT style edit box is shown on its attached KS_WITHTEXT style soft keyboard.

See Also

KM_SETDEFKEY, KM_SETPROMPT, KM_GETPROMPTLENGTH

KM_GETPROMPTLENGTH

The KM_GETPROMPTLENGTH message is sent by applications to KS_WITHTEXT style soft-keyboard controls or edit boxes to get the length of the prompt displayed in the prompt area.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value

The return value is the length of the prompt in bytes.

Comments

To display a prompt, a soft-keyboard control or an edit box must be KS_WITHTEXT style. The prompt of a KS_WITHTEXT style edit box is shown on its attached KS_WITHTEXT style soft keyboard.

See Also

KM_SETDEFKEY, KM_SETPROMPT, KM_GETPROMPT

KM_GETRECIPIENT

The KM_GETRECIPIENT message gets the handle of the recipient window of a soft-keyboard control.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value

The returned value is the handle of the recipient window.

See Also

KM_KEYDOWN, KM_KEYUP, KM_CHAR, KM_WAKEUP, KM_SETRECIPIENT

KM_GETTEXTLIMIT

The KM_GETTEXTLIMIT message retrieves the maximum number of characters the user is allowed to enter into a KS_WITHTEXT style soft-keyboard or edit-box control.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value

The return value is the maximum number of characters the user is allowed to enter.

Comments

This message is only valid for edit controls (TVEDITBOX) and soft-keyboard controls (TVKEYBOARD) using the KS_WITHTEXT style.

See Also

EM_LIMITTEXT

KM_KEYDOWN

The KM_KEYDOWN message notifies an application that a button on a soft-keyboard control has been pressed.

Parameters

WPARAM *wParam*
Specifies the virtual-key code of the button that was pressed.

LPARAM *lParam*
Specifies the repeat count in the low-order word.

Comments

The virtual-key codes specified in *wParam* are identical to the standard Windows virtual-key codes. The message sequence for soft keyboards is similar to the message sequence for IBM-style keyboards with Windows.

This message will only be posted by soft keyboards that don't have the KS_WITHTEXT style.

See Also

KM_KEYUP, KM_CHAR, WM_KEYDOWN

KM_KEYUP

The KM_KEYUP message notifies an application that a button on a soft-keyboard control has been released.

Parameters

WPARAM *wParam*
Specifies the virtual-key code of the button that was released.

LPARAM *lParam*
Specifies the repeat count in the low-order word.

Comments

The virtual-key codes specified in *wParam* are identical to the standard Windows virtual-key codes. The message sequence for soft keyboards is similar to the message sequence for IBM-style keyboards with Windows.

See Also

KM_KEYDOWN, KM_CHAR, WM_KEYUP

KM_MOVESKB

The KM_MOVESKB message changes the position of a soft-keyboard control or a keyboard control associated with an edit control.

Parameters

WPARAM *wParam*
Specifies one of the following flags to indicate whether to redraw the soft-keyboard control after moving it:

TRUE
Redraw the soft-keyboard control.

FALSE
Don't redraw the soft-keyboard control.

LPARAM *lParam*
Specifies a **POINT** structure containing the screen coordinates of the upper-left corner of the keyboard control.

Return Value

The return value is always zero.

Comments

Applications cannot change the size of a soft-keyboard control.

KM_SETDEFKEY

The KM_SETDEFKEY message sets the state of the Caps Lock button on a soft-keyboard control or edit control. It also sets the button with the initial focus.

Parameters

WPARAM *wParam*
Specifies the state of the Caps Lock button and the button with the initial focus by using a combination of the following flags:

SETDEFKEY_CAPSLOCK
Turns Caps Lock on.

SETDEFKEY_CAPSUNLOCK
Turns Caps Lock off.

SKB_0 to SKB_9
Numeric and special-character buttons.

SKB_A to SKB_Z
Alpha buttons.

SKB_DELETE
Delete button.

SKB_SPACE
Space button.

SKB_ENTER
Enter button.

SKB_ESC
Cancel button.

LPARAM *lParam*
Is not used; must be zero.

Return Value

The return value is nonzero if the request is successful; otherwise, the return value is zero.

Comments

If neither the SETDEFKEY_CAPSLOCK or SETDEFKEY_CAPSUNLOCK flags are specified, the state of the Caps Lock button is not changed. If no button indexes are specified, the button with the focus remains unchanged.

When a soft-keyboard control is created, the Caps Lock button is on and the ENTER button has the focus.

See Also

KM_SETPROMPT, KM_GETDEFKEY

KM_SETDEFTEXT

The KM_SETDEFTEXT message sets the text that appears in the text-entry area of a soft-keyboard or edit control. It also changes the size of the buffer used to store text entered with the control.

Parameters

WPARAM *wParam*
Specifies the maximum number of characters of text the soft keyboard or edit box can accommodate. If *wParam* is set to SETDEFTEXT_KEEPMAXLENGTH, the limit is not changed and this parameter is ignored.

LPARAM *lParam*
Specifies a pointer (LPSTR) to a null-terminated string that appears in the text-entry area of the control.

Return Value

The return value is zero if the message is processed successfully; otherwise, it is one of the following error codes:

SKB_ERR
The control is not an edit-box control or a soft-keyboard control using the KW_WITHTEXT style.

SKB_ERRSPACE
There is not enough memory for the requested buffer size.

Comments This message is valid only for edit-box controls (TVEDITBOX) and soft-keyboard controls (TVKEYBOARD) using the KS_WITHTEXT style. The maximum length of the buffer does not include space for the null terminator.

See Also KM_SETPROMPT, KM_SETDEFKEY, EM_LIMITTEXT

KM_SETPROMPT

The KM_SETPROMPT message is sent by applications to soft-keyboard controls or edit boxes using the KS_WITHTEXT style to set the text displayed in the prompt area.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Specifies a pointer (LPSTR) to a null-terminated string containing the prompt text.

Return Value The return value is nonzero if the request is successful; otherwise, the return value is zero.

Comments To display a prompt, a soft-keyboard control or an edit box must use the KS_WITHTEXT style. The prompt of an edit box using the KS_WITHTEXT style is shown on its associated soft-keyboard control.

See Also KM_SETDEFKEY, KM_GETPROMPT

KM_SETRECIPIENT

The KM_SETRECIPIENT message sets the window that will receive input from a soft-keyboard control.

Parameters

WPARAM *wParam*
Specifies the handle of the window to receive input from the soft keyboard.

LPARAM *lParam*
Is not used; must be zero.

Return Value

If the request is successful, the return value is nonzero; otherwise, the return value is zero.

Comments

Applications should not set the recipient to be another soft-keyboard control or an edit-box control.

When the Enter button or the Cancel button is pressed on the soft keyboard associated with an edit-box control or on a soft-keyboard control using the KS_WITHTEXT style, a WM_COMMAND message is sent to the recipient window with an EN_ENTER or EN_CANCEL notification code.

After a soft-keyboard control notifies the application of input, it remains on the screen unless it is destroyed by the application. This allows the application to prompt for additional input by changing the prompt and reawakening the keyboard. When applications receive a notification message, they must either destroy the control or reawaken it by using a KM_WAKEUP message.

Example

The following code fragment processes a WM_COMMAND message by destroying the keyboard if the Cancel button is pressed and reawakening it for additional input if the Enter button is pressed:

```
case WM_COMMAND:
    // Destroy the keyboard control on Cancel button
    if(LOWORD(lParam) == hwndKeyboard1)
    {
        if(HIWORD(lParam) == EN_CANCEL)
            PostMessage(hwndKeyboard1, WM_CLOSE, 0, 0L);

        // Reawaken the keyboard control on Enter button
        else if(HIWORD(lParam) == EN_ENTER)
        {
            ...
            GetWindowText(hwndKeyboard1, lpstrBuffer,
                          sizeof lpstrBuffer);
            ...
            PostMessage(hwndKeyboard1, KM_WAKEUP, 0, 0L);
        }
    }
    break;
```

See Also

KM_KEYDOWN, KM_KEYUP, KM_CHAR, KM_WAKEUP, KM_GETRECIPIENT

KM_WAKEUP

The KM_WAKEUP message reactivates a soft-keyboard control.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value

The return value is always zero.

Comments

Soft-keyboard controls using the KS_SYSMODAL style function like system-modal dialog boxes when activated. After either the Enter or Cancel button is pressed, the control remains inactive until it receives a KM_WAKEUP message.

LB_GETPOPUPRECT

The LB_GETPOPUPRECT message is sent by applications to a list-box control to retrieve the window rectangle of a popup list box as it will be in the activated state. This rectangle will contain screen coordinates.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Long pointer to the rectangle structure that will be filled with the popup rectangle coordinates.

Return Value

The return value is a FALSE if the pointer is not valid or the list box receiving this message is not LBS_POPUP style; otherwise, the return value is TRUE.

LB_GETSELAREA

The LB_GETSELAREA message is sent by applications to a list-box control to retrieve the offsets of the top and bottom of the selection area in the current selection rectangle of a list box.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value The return value is a DWORD containing the top and bottom coordinates relative to the top of the list-box client area. The low-order word is the top offset and the high-order word is the bottom offset.

See Also LB_SETSELAREA

LB_SETSELAREA

The LB_SETSELAREA message is sent by applications to a list-box control to set the current selection area for the list box.

Parameters WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Specifies offsets of the top and bottom of the selection area. The low-order word is the offset of the top and the high-order word is the offset of the bottom. If this value is NULL the selection rectangle assumes the default position.

Return Value The return value is TRUE if successful; otherwise, it is FALSE.

See Also LB_GETSELAREA

SBM_ENABLE_ARROWS

The SBM_ENABLE_ARROWS message is sent by applications to scroll-bar and spin-button controls to independently enable or disable the control's arrow buttons.

Parameters WPARAM *wParam*
Specifies whether to enable or disable the arrow buttons. This parameter must be one of the following values:

ESB_ENABLE_BOTH
Enables both arrow buttons.

ESB_DISABLE_LTUP
Enables the left (horizontal style) or top (vertical style) arrow button.

ESB_DISABLE_RTDN
Enables the right (horizontal style) or bottom (vertical style) arrow button.

ESB_DISABLE_BOTH
Disables both arrow buttons.

LPARAM *lParam*
Is not used; must be zero.

Return Value The return value is TRUE if the enabled state changed; otherwise, it is FALSE.

Comments This message is provided for compatibility with standard Windows scroll-bar controls. It can be used in place of the **EnableScrollBar** function.

SBM_GETCHANNELAREA

The SBM_GETCHANNELAREA message is sent by applications to scroll-bar controls to retrieve the width and position of the channel rectangle for the scroll bar.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value The return value is a DWORD specifying the width and position of the channel relative to the client rectangle of the scroll-bar control. For vertical scroll bars, the low-order word specifies the offset to the left edge of the channel and the high-order word specifies the offset to the right edge of the channel. For horizontal scroll bars, the low-order word specifies the offset to the top edge of the channel and the high-order word specifies the offset to the bottom edge of the channel.

See Also SBM_SETCHANNELAREA

SBM_SETCHANNELAREA

The SBM_SETCHANNELAREA message is sent by applications to scroll-bar controls to set the width and position of the channel rectangle for the scroll bar.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Specifies a DWORD containing the width and position of the channel relative to the client rectangle of the scroll bar. For vertical scroll bars, the low-order word specifies the offset to the left edge of the channel and the high-order word specifies the offset to the right edge of the channel. For horizontal scroll bars, the low-order word specifies the offset to the top edge of the channel and the high-order word specifies the offset to the bottom edge of the channel.

Return Value The return value is TRUE if successful; otherwise, it is FALSE.

See Also SBM_GETCHANNELAREA

SM_GETDISPLAYEXTENT

The SM_GETDISPLAYEXTENT message retrieves the display rectangle of a TV show-box control.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Specifies a pointer (LPRECT) to a **RECT** structure that will be filled with the extent of the display area, relative to the client area of the control.

Return Value The return value is nonzero if successful; otherwise, it is zero.

Comments The client area of a TV show-box control consists of the display area, a 3-D edge area, and a frame area. To ensure the display area is clipped before drawing, applications can use **IntersectClipRect** to set the clipping region to the display area returned by SM_GETDISPLAYEXTENT.

See Also SM_SETDISPLAYEXTENT

SM_SETDISPLAYEXTENT

The SM_SETDISPLAYEXTENT message sets the display area of a TV show-box control. The control window is resized and repositioned to accommodate the new display area.

Parameters

WPARAM *wParam*
Specifies one of the following flags indicating whether or not to return the new coordinates of the window rectangle for the control:

TRUE
The message returns the coordinates of the window rectangle of the TV show-box control (relative to the parent window) in the **RECT** structure specified by *lParam*.

FALSE
The message does not change the contents of the **RECT** structure specified by *lParam*.

LPARAM *lParam*
Specifies a pointer (LPRECT) to a **RECT** structure containing the new coordinates of the display area, relative to the client area of the control.

Return Value

The return value is nonzero if successful; otherwise, it is zero.

Comments

The client area of a TV show-box control consists of the display area, a 3-D edge area, and a frame area.

See Also

SM_GETDISPLAYEXTENT

WM_GETBITMAP

The WM_GETBITMAP message is sent by applications to retrieve information about a custom bitmap for a given control.

Parameters

WPARAM *wParam*
Specifies options for getting the bitmap information using a combination of the following flags:

GETBITMAP_BTNUP
Get information about the bitmap associated with the up state of a button control.

GETBITMAP_BTNDOWN
Get information about the bitmap associated with the down state of a button control.

GETBITMAP_BTNFACE
Get information about the bitmap associated with the text area of a button control.

GETBITMAP_SHOWBOX
Get information about the bitmap associated with the display area of a show-box control.

GETBITMAP_ALL
Get the bitmap, background mode, and background color. The *lParam* parameter specifies a far pointer (LPTV_CTLBITMAP or LPTV_FACEBITMAP) to an **TV_CTLBITMAP** or **TV_FACEBITMAP** structure.

GETBITMAP_HBMONLY
Get only the bitmap. The *lParam* parameter specifies a far pointer to a bitmap handle (HBITMAP FAR *).

GETBITMAP_BKMODEONLY
Get only the background mode. The *lParam* parameter specifies a far pointer to an integer (int FAR *).

GETBITMAP_COLORONLY
Get only the background color. The *lParam* parameter specifies a far pointer to a COLORREF (COLORREF FAR *).

LPARAM *lParam*
The value of *lParam* depends on the flags specified for *wParam*. See the descriptions of the GETBITMAP_ALL, GETBITMAP_HBMONLY, GETBITMAP_BKMODEONLY, and GETBITMAP_COLORONLY flags for the value of *lParam*.

Comments

This message should only be sent to controls for which an application has already set a custom bitmap using WM_GETBITMAP. The only controls that recognize this message are the button, show-box, and scroll-bar controls.

For button controls, applications must specify either the GETBITMAP_BTNUP, GETBITMAP_BTNDOWN, or GETBITMAP_BTNFACE flag to indicate which of the control's bitmaps to retrieve.

For show-box controls, applications must specify the GETBITMAP_BTNFACE flag to retrieve the bitmap for the face of the control.

For scroll-bar controls, the retrieved bitmap is for the thumb of the scroll-bar.

In addition, applications must specify either the GETBITMAP_HBMONLY, GETBITMAP_BKMODEONLY, GETBITMAP_COLORONLY, or GETBITMAP_ALL flag to indicate whether to retrieve the bitmap, the background mode, the background color, or all of these attributes.

See Also

WM_SETBITMAP, **TV_CTLBITMAP**, **TV_FACEBITMAP**

WM_GETCOLOR

The WM_GETCOLOR message is sent by applications to retrieve information about the colors used in drawing a given control.

Parameters

WPARAM *wParam*

Specifies options for getting the color information using a combination of the following flags:

GETCOLOR_ALL

Get the color of all parts of the control element. The *lParam* parameter specifies a far pointer (LPTV_CTLCOLOR) to a **TV_CTLCOLOR** structure.

GETCOLOR_HIGHLIGHT

Get the highlight color. The *lParam* parameter specifies a far pointer to a COLORREF (COLORREF FAR *).

GETCOLOR_FRAME

Get the frame color. The *lParam* parameter specifies a far pointer to a COLORREF (COLORREF FAR *).

GETCOLOR_FACE

Get the face color. The *lParam* parameter specifies a far pointer to a COLORREF (COLORREF FAR *).

GETCOLOR_SHADOW

Get the shadow color. The *lParam* parameter specifies a far pointer to a COLORREF (COLORREF FAR *).

GETCOLOR_TEXT

Get the text color. The *lParam* parameter specifies a far pointer to a COLORREF (COLORREF FAR *).

GETCOLOR_SBTHUMB

Get the color of the thumb element of a scroll-bar control.

GETCOLOR_SBCHANNEL

Get the color of the channel element of a scroll-bar control.

Comments

The GETCOLOR_ALL, GETCOLOR_HIGHLIGHT, GETCOLOR_FRAME, GETCOLOR_FACE, GETCOLOR_SHADOW, and GETCOLOR_TEXT flags are mutually exclusive—applications must specify one of these flags to indicate a specific part of the control element.

The GETCOLOR_SBTHUMB and GETCOLOR_SBCHANNEL flags are mutually exclusive and should only be specified when retrieving colors for the thumb or channel element of a scroll-bar control. Only the highlight, face, and shadow colors of the thumb element can be retrieved. Only the highlight, face, shadow, and scroll arrow colors of the channel element can be retrieved. To retrieve the color of scroll arrows, specify the GETCOLOR_TEXT flag.

See Also WM_SETCOLOR, **TV_CTLCOLOR**

WM_QUERYFOCUS

The WM_QUERYFOCUS message is sent by applications to retrieve information about the sub-focus of a compound control other than a soft-keyboard control. To retrieve the sub-focus information of a soft-keyboard control, use the KM_GETDEFKEY message.

Parameters

WPARAM *wParam*
Is not used; must be zero.

LPARAM *lParam*
Is not used; must be zero.

Return Value The return value is one of the following values:

QF_LEFT
The left button has the focus.

QF_RIGHT
The right button has the focus.

QF_UP
The up button has the focus.

QF_DOWN
The down button has the focus.

See Also KM_SETDEFKEY, KM_GETDEFKEY

WM_SETBITMAP

The WM_SETBITMAP message is sent by applications to specify custom bitmaps for a given control. An application can set bitmaps for the up (or unchecked) state, the down (or checked) state, or for the text area of a button. In addition, an application can set the background mode and background color for a custom bitmap.

Parameters

WPARAM *wParam*

Specifies options for setting the bitmap using a combination of the following flags:

SETBITMAP_BTNUP

The given bitmap, background mode, or background color is for the up state of a button control.

SETBITMAP_BTNDOWN

The given bitmap, background mode, or background color is for the down state of a button control.

SETBITMAP_BTNFACE

The given bitmap, background mode, or background color is for the text area of a button control.

SETBITMAP_SHOWBOX

The given bitmap, background mode, or background color is for the display area of a show-box control.

SETBITMAP_ALL

Set the bitmap, background mode, and background color. The *lParam* parameter specifies a far pointer (LPTV_CTLBITMAP or LPTV_FACEBITMAP) to an **TV_CTLBITMAP** or **TV_FACEBITMAP** structure.

SETBITMAP_HBMONLY

Set only the bitmap. The low-order word of the *lParam* parameter specifies a handle (HBITMAP) to the bitmap.

SETBITMAP_BKMODEONLY

Set only the background mode used in rendering the bitmap. The low-order word of the *lParam* parameter specifies the background mode.

SETBITMAP_COLORONLY

Set only the background color used in rendering the bitmap. The *lParam* parameter specifies a COLORREF for the background color.

LPARAM *lParam*

The value of *lParam* depends on the flags specified for *wParam*. See the descriptions of the SETBITMAP_ALL, SETBITMAP_HBMONLY, SETBITMAP_BKMODEONLY, and SETBITMAP_COLORONLY flags for the value of *lParam*.

Comments

The only controls that recognize this message are button, show-box, and scroll-bar controls.

For button controls, applications must specify either the SETBITMAP_BTNUP, SETBITMAP_BTNDOWN, or SETBITMAP_BTNFACE flag to indicate which of the control's bitmaps to set. The SETBITMAP_BTNUP and SETBITMAP_BTNDOWN flags indicate the given bitmap is to be used to represent the button in either the normal or pressed state. The SETBITMAP_BTNFACE flag indicates the given bitmap is to be placed on the face of the button.

For show-box controls, applications must specify the SETBITMAP_BTNFACE flag to indicate the bitmap is for the face of the control.

For scroll-bar controls, the given bitmap is assumed to be for the thumb of the scroll-bar.

In addition, applications must specify either the SETBITMAP_HBMONLY, SETBITMAP_BKMODEONLY, SETBITMAP_COLORONLY, or SETBITMAP_ALL flag to indicate whether to set the bitmap, the background mode, the background color, or all of these attributes.

Applications are responsible for deleting the bitmap objects that are set using WM_SETBITMAP. A good time to do this is when a WM_DESTROY message is received. Bitmaps must be 16-color device-independent bitmaps.

See Also

WM_GETBITMAP, **TV_CTLBITMAP**, **TV_FACEBITMAP**

WM_SETCOLOR

The WM_SETCOLOR message is sent by applications to change the colors used in drawing a control.

Parameters

WPARAM *wParam*

Specifies options for setting the color using a combination of the following flags:

SETCOLOR_ALL

Set the color of all parts of the control element. The *lParam* parameter specifies a far pointer (LPTV_CTLCOLOR) to a **TV_CTLCOLOR** structure.

SETCOLOR_HIGHLIGHT

Set only the highlight color. The *lParam* parameter specifies a COLORREF.

SETCOLOR_FRAME
Set only the frame color. The *lParam* parameter specifies a COLORREF.

SETCOLOR_FACE
Set only the face color. The *lParam* parameter specifies a COLORREF.

SETCOLOR_SHADOW
Set only the shadow color. The *lParam* parameter specifies a COLORREF.

SETCOLOR_TEXT
Set only the text color. The *lParam* parameter specifies a COLORREF.

SETCOLOR_SBTHUMB
Set the color of the thumb element of a scroll-bar control.

SETCOLOR_SBCHANNEL
Set the color of the channel element of a scroll-bar control.

LPARAM *lParam*
The value of *lParam* depends on the flags specified for *wParam*. See the descriptions of the SETCOLOR_ALL, SETCOLOR_HIGHLIGHT, SETCOLOR_FRAME, SETCOLOR_FACE, SETCOLOR_SHADOW, and SETCOLOR_TEXT flags for the value of *lParam*.

Comments

The SETCOLOR_ALL, SETCOLOR_HIGHLIGHT, SETCOLOR_FRAME, SETCOLOR_FACE, SETCOLOR_SHADOW, and SETCOLOR_TEXT flags are mutually exclusive—applications must specify one of these flags to indicate which part of the control element to change.

The SETCOLOR_SBTHUMB and SETCOLOR_SBCHANNEL flags are mutually exclusive and should only be specified when setting colors for the thumb or channel element of a scroll-bar control. Only the highlight, face, and shadow parts of the thumb element can be changed. Only the highlight, face, shadow, and scroll arrows of the channel element can be changed. To set the color of scroll arrows, specify the SETCOLOR_TEXT flag.

See Also

WM_GETCOLOR, **TV_CTLCOLOR**

CHAPTER 7

Data Structures

This chapter describes data structures used in the Modular Windows API. It contains the following sections:

- An overview of new Modular Windows data structures. This overview includes brief descriptions of the data structures.
- Detailed descriptions of the new data structures, organized alphabetically. These descriptions include the structure definition and the type and a description of the contents of each of the fields in the structure.

Data Structure Overview

The following table lists the Modular Windows data structures along with a brief description of each structure:

Data Structure	Description
DIRVECTORS	Contains focus vectors for the focus manager.
TV_CTLBITMAP	Contains information about a custom bitmap used in a control.
TV_CTLCOLOR	Contains color information for a control.
TV_FACEBITMAP	Contains information about a bitmap for the face of a TV button control.

Data Structure Reference

This section lists the Modular Windows data structures alphabetically. Each entry includes a description of the structure, the type definition of the structure, a description of each of the fields of the structure, and optional comments, examples, and cross-references.

DIRVECTORS

The **DIRVECTORS** structure specifies focus vectors. Focus vectors determine which control gets the focus when the focus is moved.

```
typedef struct {
  hwnd up;
  hwnd down;
  hwnd left;
  hwnd right;
}DIRVECTORS;
```

Fields

up
Specifies the control that will get the focus if the focus is moved up.

down
Specifies the control that will get the focus if the focus is moved down.

left
Specifies the control that will get the focus if the focus is moved left.

right
Specifies the control that will get the focus if the focus is moved right.

See Also **fmSetWindowVectors, fmGetWindowVectors**

TV_CTLBITMAP

The **TV_CTLBITMAP** structure contains information about a custom bitmap associated with a control.

```
typedef struct {
  HBITMAP hBmp;
  int nBmpBkMode;
  COLORREF crBmpBk;
}TV_CTLBITMAP;
```

Fields

hBmp
Specifies a handle to the bitmap.

nBmpBkMode
Specifies the background mode for rendering the bitmap using one of the following flags:

OPAQUE
Background is filled with current background color before rendering the bitmap.

TRANSPARENT
Background is not changed before rendering the bitmap.

NEWTRANSPARENT
Pixels in the bitmap that are the same color as the background color are not drawn when rendering the bitmap.

crBmpBk
Specifies the bitmap background color.

See Also WM_SETBITMAP, WM_GETBITMAP

TV_CTLCOLOR

The **TV_CTLCOLOR** structure contains information about the colors used to render a control.

```
typedef struct {
  COLORREF Face;
  COLORREF Highlight;
  COLORREF Shadow;
  COLORREF Frame;
  COLORREF Text;
}TV_CTLCOLOR;
```

Fields

Face
Specifies the face color.

Highlight
Specifies the highlight color.

Shadow
Specifies the shadow color.

Frame
Specifies the frame color.

Text
Specifies the text color (or scroll-arrow color for the channel element of a scroll-bar control).

See Also WM_SETCOLOR, WM_GETCOLOR

TV_FACEBITMAP

The **TV_FACEBITMAP** structure contains information about a bitmap associated with the face area of a TV button control or with a TV show-box control.

```
typedef struct {
  TV_CTLBITMAP tvBmpInfo;
  int nBmpTextSeparation;
  int nBmpAlignOffset;
  BYTE byAlignStyle;
}TV_FACEBITMAP;
```

Fields

tvBmpInfo
Specifies information about the bitmap, including the background mode and background color for rendering the bitmap.

nBmpTextSeparation
Specifies the separation in pixels between the text and the bitmap.

nBmpAlignOffset
Specifies the offset in pixels from the edge of the control face to the bitmap. With button controls, this field is used only when the **byAlignStyle** field is set to either BMS_LEFTALIGNBITMAP or BMS_RIGHTALIGNBITMAP. With show-box controls, this field is used only when the **byAlignStyle** field is set to either BMS_LEFTALIGNBITMAP, BMS_RIGHTALIGNBITMAP, BMS_TOPBITMAP, or BMS_BOTTOMBITMAP.

byAlignStyle
Specifies the alignment of the bitmap and the text using one of the following style flags:

BMS_LEFTBITMAP
Draw the bitmap to the left of the text. Center the text and bitmap in the face of the control.

BMS_RIGHTBITMAP
Draw the bitmap to the right of the text. Center the text and bitmap in the face of the control.

BMS_CENTERBITMAP
Draw the bitmap in the center of the control and overlay the text over the top of the bitmap.

BMS_LEFTALIGNBITMAP
Draw the bitmap to the left of the text and align it to the left of the control.

BMS_RIGHTALIGNBITMAP
Draw the bitmap to the right of the text and align it to the right of the control.

BMS_TOPBITMAP
Align the bitmap with the top of the control. This alignment style is valid only with show-box controls.

BMS_BOTTOMBITMAP
Align the bitmap with the bottom of the control. This alignment style is valid only with show-box controls.

See Also WM_SETBITMAP, WM_GETBITMAP, **TV_CTLBITMAP**

CHAPTER 8

File Formats

This chapter provides details about new Modular Windows file formats required for TV-based multimedia players. All of the new file formats are for device-independent bitmaps (DIBs). The formats detailed in this chapter are extensions to the Windows DIB format documented in the *Programmer's Reference, Volume 4: Resources* manual in the Windows Software Development Kit.

For details on DIB file formats specific to VIS players, see Appendix B, "VIS Programming Notes."

RGB DIB Formats

Efficient utilization of the new video modes provided by TV-based players requires a new format to accommodate 16-bit RGB DIBs. Standard 8-bit DIBs (256 colors) use a color table to encode the color information. The new 16-bit RGB DIBs don't have a color table, but encode the color information directly into the 16 bits representing each pixel. There are two types of 16-bit RGB DIBs:

- RGB555—32K colors using five bits each for red, green, and blue.
- RGB565—64K colors using five bits each for red and blue, and six bits for green.

BITMAPINFOHEADER Structure for RGB555 and RGB565 DIBs

The following table contains information on the fields of the **BITMAPINFOHEADER** structure for RGB555 and RGB565 DIBs:

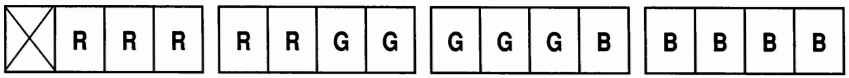
Field	Description
biSize	Size in bytes of the **BITMAPINFOHEADER** structure.
biWidth	Width of the bitmap in pixels.
biHeight	Height of the bitmap in pixels.
biPlanes	Set to 1.
biBitCount	Set to 16.
biCompression	For RGB555, set to 0 (BI_RGB). For RGB565, set to the four-character code 'R565'.
biSizeImage	Size in bytes of the image.
biXPelsPerMeter	Horizontal resolution in pixels per meter.
biYPelsPerMeter	Vertical resolution in pixels per meter.
biClrUsed	Set to 0.
biClrImportant	Set to 0.

The following code fragment shows how to create the four-character code required in the **biCompression** field for RGB565 DIBs:

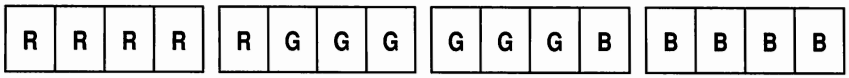
```
#include <mmsystem.h>
...

bmih.biCompression = mmioFOURCC('R', '5', '6', '5');
```

RGB555 and RGB565 Pixel Encoding

The following illustrations show the pixel encoding for RGB555 and RGB565 DIBs:



Pixel encoding for RGB555 DIB



Pixel encoding for RGB565 DIB

CHAPTER 9

Tools

The Microsoft Modular Windows Software Developer's Kit (SDK) includes tools to help you develop your application on PC-based development systems and debug and test it on TV-based multimedia players. This chapter describes these tools and provides information on using each one.

The Modular Windows SDK includes the following tools:

- Microsoft Transport Layer TSR (TLTSR.EXE), a tool providing serial communication between a PC and a TV-based player.
- Microsoft Redirected File Server (RFSERVER.EXE), a tool that works in conjunction with the Transport Layer TSR to provide redirected file services on a TV-based player. Redirected file services let you use files residing on a host PC's hard disk to replace or supplement files on the player's CD-ROM disc.
- Microsoft Modular Windows 80286 Debugger (WDEB286.EXE), a debugger for applications running on TV-based players using the 80286 microprocessor.
- NoEcho (NOECHO.EXE), a tool that works in conjunction with the Microsoft Redirected File Server to display debugging messages generated by an application running on a TV-based player.
- Microsoft Modular Windows Heap Walker (MODWHEAP.EXE), a tool that lets you examine the local and global heaps used by applications and dynamic-link libraries (DLLs) running under Modular Windows.
- Microsoft MS-DOS Monitor (DOSMON.COM), a tool that reports all calls to unsupported MS-DOS functions.
- Microsoft Color Table Converter (CLTCONV.EXE), a tool that converts bitmap and icon color tables.
- Conver24 (CONVER24.EXE), an application for Windows that converts images to new file formats supported by Modular Windows.

Debugging Applications on a TV-Based Player

After testing and debugging your application on a development system, you can use the tools provided in the Modular Windows SDK to test and debug the application on a TV-based player. The SDK debugging tools let you perform the following tasks:

- Install breakpoints, view the contents of registers and variables, and perform other debugging operations
- View debugging messages generated by your application
- Replace and supplement files on the CD-ROM disc with files residing on a PC

The following sections discuss how to use the debugging tools to debug applications running on a TV-based player. For a discussion on overall debugging strategy see Chapter 3, "Creating Applications for Modular Windows," in the *Getting Started* manual.

Hardware Requirements for Debugging

To debug applications running on a TV-based player, you'll need the following hardware:

- A TV-based player (the target).
- A PC running MS-DOS version 3.22 or later (the host).
- Approximately 7K of free RAM on the TV-based player. An additional 80K of free RAM is required to run the Modular Windows debugger.
- A serial interface for the TV-based player. Consult the manufacturer of the TV-based player for information about obtaining this serial interface.
- A serial null-modem cable. The cable connects the COM1 port on your host PC to the serial interface on the TV-based player.

In addition, you'll need a CD-ROM disc containing a current version of your application.

About the Transport Layer and File Redirection

The debugging tools for applications running on TV-based players depend on a *transport layer* that provides a serial communications link between the player and the host PC. The transport layer is provided by ROM-based software in the player and the Microsoft Transport Layer TSR utility (TLTSR.EXE) running on the host. The serial link operates at a baud rate of 115K.

The transport layer, in conjunction with the Microsoft Redirected File Server utility (RFSERVER.EXE), supports *file redirection*, diverting file operations from the player's CD-ROM disc to a hard disk on the host PC. File redirection allows you to test, debug, and expand your application without creating a new CD every time a file is changed or added to the application. You can redirect executable files as well as data files.

Starting the Transport Layer

You must start the transport layer before using file redirection, the Modular Windows 80286 Debugger, or the NoEcho utility.

▶ **To start the transport layer:**

1. Run the Transport Layer TSR (TLTSR.EXE) on the host PC.
2. Run the Redirected File Server utility (RFSERVER.EXE) on the host PC.
3. Restart the target TV-based player using your application disc.

You must run the Transport Layer TSR before running the Redirected File Server utility.

Using the Transport Layer TSR Tool

The Transport Layer TSR is a terminate-stay-resident (TSR) program that activates the Modular Windows transport layer, allowing serial communication between the TV-based player and the host PC. To start the Transport Layer TSR, use the **tltsr** command.

Command-Line Syntax

The following line shows the **tltsr** command-line syntax:

tltsr

There are no arguments to the **tltsr** command.

Note Once started, the Transport Layer TSR remains active through the duration of the MS-DOS session. Don't run it more than once in a single MS-DOS session. To determine if the Transport Layer TSR is currently running, use the **mem** command with the /**c** option.

Using the Redirected File Server Tool

The Redirected File Server tool works in conjunction with the Transport Layer TSR to provide file-redirection services. To start the Redirected File Server, use the **rfserver** command.

Command-Line Syntax

The following line shows the **rfserver** command-line syntax:

rfserver [*options*] *directory*

The *directory* argument identifies the directory on the host PC containing files you want to redirect to the CD-ROM application.

Specifying Options

The **rfserver** command's *options* argument can include one or more of the following options:

-n
Runs the Redirected File Server in non-TSR mode. If this option is omitted, the program runs as a TSR. The performance of file redirection is improved in non-TSR mode.

-m
Displays status messages when redirected files are opened, read, and closed. These messages are useful for monitoring the use of redirected files and for verifying that redirection is working correctly.

To view debug messages generated by your application or to run the Modular Windows 80286 Debugger, you must run the Redirected File Server as a TSR. If you are running as a TSR, you must restart the TV-based player if you change any of the redirected files.

Note Once started as a TSR, the Redirected File Server remains active through the duration of the MS-DOS session. Don't run it more than once in a single MS-DOS session. To determine if the Redirected File Server is currently running, use the **mem** command with the **/c** option.

Tips for Using File Redirection

The following tips will help you get the most benefit from the Modular Windows file-redirection tools:

- Start with the current version of your application on a CD-ROM disc. Use file redirection to update files that are already on the disc, or to add new files.
- Because data transfer across a serial link is not as fast as data transfer from a CD-ROM, you might notice a decrease in your application's performance using file redirection. To minimize the performance degradation, don't redirect files such as audio and video files that require high-speed data transfer.
- Don't run protected-mode software, such as Windows or network software, while you're using the transport layer for file redirection or debugging.
- If the TV-based player hangs during file redirection, restart it. Redirection should resume when the TV-based player has been restarted.
- Always keep a disc in the CD-ROM drive, even if all of your files are redirected. The CD-ROM device driver requires that a disc be in the drive.

Using the NoEcho Utility to View Debug Messages

The NoEcho utility (NOECHO.EXE) lets you view debug messages generated by your application using the **OutputDebugString** function. To run the NoEcho utility, use the **noecho** command. Before running NoEcho, you must start the Transport Layer TSR. NoEcho will run in conjunction with the Redirected File Server if the Redirected File Server is run as a TSR.

Command-Line Syntax

The following line shows the **noecho** command-line syntax:

noecho [*options*]

Specifying Options

The **noecho** command's *options* argument can include one or more of the following options:

-f *filename*
Enables logging; sends output to the file specified by *filename*.

-a
If logging is enabled, logs all output from the Debugger as well as from the **OutputDebugString** function.

-?
Displays a list of the **noecho** command-line arguments.

To exit NoEcho, press the ESC key on the host PC's keyboard.

Using Modular Windows 80286 Debugger

The Modular Windows 80286 Debugger (WDEB286.EXE) lets you test and debug applications and dynamic-link libraries running under Modular Windows on a TV-based player. You can inspect and manipulate test code and environment status, install breakpoints, and perform other debugging operations. Modular Windows 80286 Debugger is similar to Windows 80386 Debugger. You should be familiar with 80386 Debugger before using 80286 Debugger.

This chapter provides details about starting the debugger and suggestions about using it with TV-based players, but does not cover all of the debugger's options and commands. For complete information about all of the options and commands, see the information on the 80386 Debugger in the *Programming Tools* manual in the Microsoft Windows Software Development Kit.

Note Commands requiring an 80386 microprocessor, such as hardware breakpoints, are not supported by 80286 Debugger.

Starting 80286 Debugger

Before starting 80286 Debugger, you must start the transport layer to allow communications between the TV-based player and the host PC. See "Starting the Transport Layer," earlier in this chapter, for details on starting the transport layer.

To start 80286 Debugger, you must edit the launch file and replace the line containing the **modwin** command with a line containing the **wdeb286** command. See Chapter 3, "Creating Applications for Modular Windows," in the *Getting Started* manual for details on the launch file.

Command-Line Syntax

The following line shows the command-line syntax for starting 80286 Debugger on a TV-based player:

wdeb286.exe [options] **gorom.com** *windir* [*environment1*, *environment2*, ...]

80286 Debugger runs the **gorom** command, which starts Modular Windows running under the debugger. The *windir* and *environment* arguments for **gorom** are identical to the arguments for **modwin**: *windir* specifies the directory containing the SYSTEM.INI file and *environment* specifies optional environment variables that the application can use.

Specifying Options

The **wdeb286** command's *options* argument can include any of the options for 80386 Debugger. In addition, the **wdeb286** command accepts the following option:

-i

Specifies the GOROM.COM program file is redirected. You should always specify this option when running 80286 Debugger.

Tips for Using 80286 Debugger

The following are suggestions for using the 80286 Debugger to debug applications running on a TV-based player:

- You can use file redirection for all of the debugging tools by placing the program files in your file-redirection directory (specified with the **rfserver** command). You can also use file redirection for the launch file to change the **wdeb286** command line without making a new CD-ROM disc.
- If you encounter INT 3 breakpoints while the debugger is starting Modular Windows, use the **g** command to proceed.
- You can enter the debugger by pressing CTRL+ALT+SYSRQ on the TV-based player's keyboard (if installed) or CTRL+C on the host PC's keyboard. When you see the debugging prompt, you can enter debugging commands.

Using Modular Windows Heap Walker

Modular Windows Heap Walker (MODWHEAP.EXE) lets you examine the local and global heaps used by applications and dynamic-link libraries. Modular Windows Heap Walker provides the same functionality as the version of Heap Walker for Windows 3.1. For complete information about Heap Walker for Windows 3.1, see the *Programming Tools* manual in the Microsoft Windows SDK.

This section describes differences you'll encounter when starting, running, and using Modular Windows Heap Walker.

Changes in Appearance

The appearance of Modular Windows Heap Walker differs from Heap Walker for Windows 3.1. The changes include the following:

- The menu bar has been replaced by a button bar. Menu items appear in the same order as in Heap Walker for Windows. When you select a button, Modular Windows displays a column of buttons, similar to the drop-down menus in Microsoft Windows 3.1.
- User-interface controls in the Heap Walker windows have been replaced by Modular Windows user-interface controls. For example, list boxes in the Main window, the Global window, and the Local Walk window have been changed to TV list boxes.

Changes in Functionality

The functionality of Modular Windows Heap Walker is very similar to Heap Walker for Microsoft Windows 3.1. The exceptions are as follows:

- Set Output Destination has been added to the File menu. This command lets you choose either the COM port or the hard disk as the destination when you choose the Save command. Once you select COM port or hard disk, any future saves in the same session are automatically made to the same location. The default setting is Hard Disk.
- Add Menu has been removed from the Main menu. It required multiple-selection list boxes, which are not available under Modular Windows.

Running Heap Walker on a TV-Based Player

Because Modular Windows doesn't provide a shell application such as Program Manager, you must consider the following guidelines when running Heap Walker with your application on a TV-based player:

- Use the **WinExec** function in your application to launch Heap Walker.
- If your application's window covers the Heap Walker menu, you must provide a way to set the focus to Heap Walker. Using the optional mouse on the player might be helpful in changing the focus between applications.

Using the MS-DOS Monitor

The MS-DOS Monitor utility (DOSMON.COM) can be used to determine if an application makes calls to MS-DOS functions not available in the Modular Windows ROM. If your application uses libraries such as a standard C-run-time library, the library might be making calls to MS-DOS you are not aware of. The MS-DOS Monitor utility will detect these unsupported calls.

MS-DOS Monitor is a TSR program that reports the function number of unsupported MS-DOS functions and the filenames of files executed or opened. MS-DOS Monitor sends output to the COM1 port using the settings of 9600 baud, no parity, 8 bits per character, and 1 stop bit. To use MS-DOS Monitor with Modular Windows applications, start MS-DOS Monitor before starting Modular Windows.

Command-Line Syntax

To run MS-DOS Monitor, use the **dosmon** command. The following line shows the command-line syntax for **dosmon**:

dosmon [*options*]

Specifying Options

The **dosmon** command's *options* argument can include one or more of the following options:

-?
Displays a list of **dosmon** command-line arguments.

/d
Exits MS-DOS Monitor and removes the TSR from memory.

Note Using a serial mouse on the COM1 port will cause a conflict with MS-DOS Monitor. Try changing your mouse to use COM2, or use a bus mouse.

Using the Color Table Converter

The Color Table Converter (CLTCONV.EXE) is an MS-DOS utility that changes the RGB values in the color table of icons, 16-color bitmaps, and 256-color bitmaps with identity palettes. It's designed to help solve color-matching anomalies that can occur when icons and bitmaps created under Microsoft Windows 3.1 are displayed by applications running under Modular Windows. For more information on color-matching anomalies, see Chapter 3, "Video Services."

The Color Table Converter utility takes a reference color table and a bitmap as input files. The reference color table contains RGB values for one or more color tables. If one of the reference color tables matches the color table in the bitmap, the color table in the bitmap is converted to use the RGB values for Modular Windows. The following list summarizes how Color Table Converter operates with various types of input files:

Input File Type	Color Table Converter Operation
16-color bitmap	Converts all 16 colors in the color table
256-color bitmap	Converts the first 10 and last 10 entries in the palette
16-color icon	Converts all 16 colors in the color table of each icon in the icon resource

Command-Line Syntax

To run the Color Table Converter utility, use the **cltconv** command. The following line shows the command-line syntax of **cltconv**:

cltconv [*options*] [-t *reference-file*] *input-file* [*output-file*]

If you don't specify an output file, the **cltconv** command will overwrite the input file with the converted output file.

Specifying Options

The **cltconv** command's options argument can include one or more of the following options:

-?
Displays a list of **cltconv** command-line arguments.

-v
Displays messages that report the progress of **cltconv**.

-t *reference-file*
Provides a reference color table. If no reference color table is provided, Color Table Converter searches the current directory and the directories specified in the PATH environment variable for a reference file named CLTCONV.TBL.

About the Reference Color-Table File

The reference color-table file is an ASCII text file containing one or more reference color tables. There are two types of reference color tables: one with 16 entries for 16-color bitmaps and icons, and one with 20 entries for 256-color bitmaps. The token **16-colors** identifies a color table with 16 entries and the token **256-colors** identifies a color table with 20 entries.

Example Color Table for 16-Color Bitmaps and Icons

The following is an example of a 16-color reference color table:

```
16-colors   (MCGA256)
      0,   0,   0   // black
    128,   0,   0   // dark red
      0, 128,   0   // dark green
    128, 128,   0   // mustard
      0,   0, 128   // dark blue
    128,   0, 128   // purple
      0, 128, 128   // dark turquoise
    192, 192, 192   // gray
    128, 128, 128   // dark gray
    255,   0,   0   // red
      0, 255,   0   // green
    255, 255,   0   // yellow
      0,   0, 255   // blue
    255,   0, 255   // magenta
      0, 255, 255   // cyan
    255, 255, 255   // white
```

Any text following the RGB values in each line is ignored. There can be no blank lines in the reference color table.

Example Color Table for 256-Color Bitmaps

The color table for 256-color bitmaps has a total of 20 entries for the first 10 and last 10 entries in the bitmap's palette. The following is an example of a reference color table for a 256-color bitmap:

```
256-colors  (MCGA256)
      0,   0,   0   // black
    128,   0,   0   // dark red
      0, 128,   0   // dark green
    128, 128,   0   // mustard
      0,   0, 128   // dark blue
    128,   0, 128   // purple
      0, 128, 128   // dark turquoise
    192, 192, 192   // gray
    192, 220, 192   // money green
    166, 202, 240   // new blue
    255, 251, 240   // off-white
    160, 160, 164   // med-gray
    128, 128, 128   // dark gray
    255,   0,   0   // red
      0, 255,   0   // green
    255, 255,   0   // yellow
      0,   0, 255   // blue
    255,   0, 255   // magenta
      0, 255, 255   // cyan
    255, 255, 255   // white
```

The Conver24 Utility

Conver24 (CONVER24.EXE) is a Windows application that converts and scales 24-bit images to new file formats used by the **DisplayDib** and **DisplayDibEx** functions.

Conver24 also has limited digital-filtering capabilities. There are several applications available specifically designed for digital filtering. The Modular Windows SDK includes a sample set of filter parameters. This section includes information about the sample parameters and a general discussion of the filtering capabilities of Conver24.

About Digital Filtering

This section does not attempt to explain digital filtering techniques or video image technology. The following publications might be helpful for information on these topics:

Elliott, Douglas F. *Handbook of Digital Signal Processing*. San Diego: Academic Press, Inc., 1987.

Embree, Paul M., and Bruce Kimble. *C Language Algorithms For Digital Signal Processing*. Englewood Cliffs: Prentice-Hall, Inc., 1991.

Gonzalez, Rafael C., and Paul Wintz. *Digital Image Processing*. Menlo Park: Addison-Wesley Publishing Company, 1987.

Oppenheim, Alan V., and Ronald W. Schafer. *Digital Signal Processing*. Englewood Cliffs: Prentice-Hall, Inc., 1975.

Rabiner, Lawrence R., and Bernard Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs: Prentice-Hall, Inc., 1975.

About Conver24

Conver24 is designed to reduce some of the annoying effects created when images are displayed on televisions, such as flicker of thin horizontal lines and the chroma-crawl phenomenon on the boundaries of vertical color bands. Conver24 is script-driven and designed to batch process multiple images.

Filtering Capabilities

Conver24 performs one-dimensional, adaptive filtering of 24-bit RGB images in the YIQ color space. Separate passes are made for the horizontal and vertical dimensions. The adaptiveness of the filtering is realized by the use of a specified threshold: convolutions are done on the color components of two adjacent pixels only if their difference is greater than the threshold value. The program allows one-dimensional convolution in the horizontal or vertical dimension with specified convolution weights.

Adaptive low-pass filtering in the vertical direction reduces line flicker, while adaptive low-pass filtering in the horizontal direction reduces chroma crawl. Thresholding the filtering eliminates unnecessary filtering on regions of slow transitions, which don't exhibit flicker or chroma crawl.

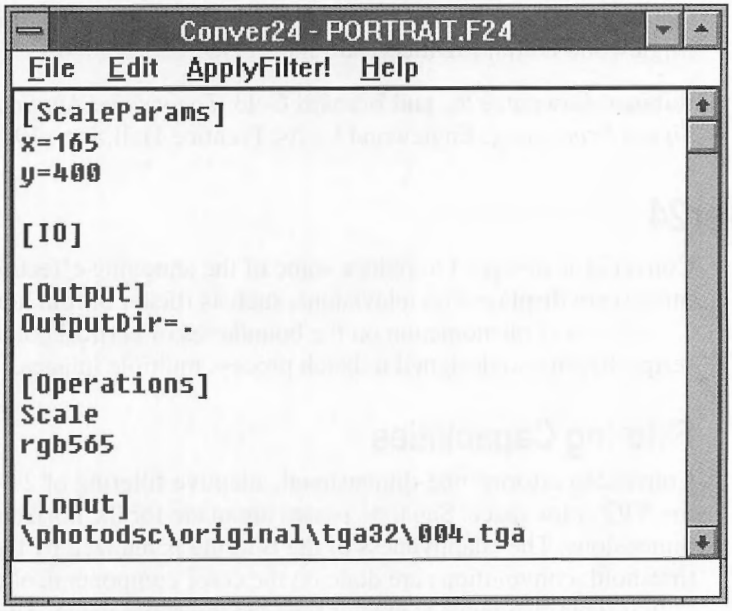
Note Conver24 can perform filtering only on 16-bit, 24-bit, and 32-bit images.

Image Scaling with Conver24

Conver24 is capable of scaling images using interpolation. The scaling routine determines the RGB value for each pixel in the scaled image by taking the weighted sum of the RGB values of the old pixels that overlap with the new pixel—the weights are proportional to the area of overlap for each old pixel. Scaling can be absolute with both x- and y-dimensions specified, or proportional with one dimension specified and the other dimension calculated by Conver24 to maintain the aspect ratio of the image.

Using Conver24

The following is the main window of Conver24:



Conver24 lets you enter a new script or load an existing one. To run the filter script, choose ApplyFilter! on the menu bar. The I/O section in the filter script is required—Conver24 will not prompt for filenames.

A Sample Low-Pass Filter

The following is an example of a script for a low-pass filter for processing images for display on televisions:

```
; NTSC filter parameter section (required only if filtering images)
[NTSCParams]

; Define filters
[filter 1]
xdim = 1
ydim = 3
weights = .1 .8 .1

[filter 2]
xdim = 1
ydim = 5
weights = .05 .15 .6 .15 .05

; Define chroma-crawl reduction filters
[ChromaCrawlReduction]
Y FilterIndex = 1
Y FilterOption = CUMULATIVE
Y Threshold = 60
Y FilterMaxScan = 2

I FilterIndex = 2
I FilterOption = CUMULATIVE
I Threshold = 12
I FilterMaxScan = 0

Q FilterIndex = 2
Q FilterOption = CUMULATIVE
Q Threshold = 20
Q FilterMaxScan = 0

; Define flicker-reduction filters
[FlickerReduction]
Y FilterIndex = 2
Y FilterOption = CUMULATIVE
Y Threshold = 70
Y FilterMaxScan = 1

I FilterIndex = 2
I FilterOption = CUMULATIVE
I Threshold = 20
I FilterMaxScan = 1
```

```
Q FilterIndex = 2
Q FilterOption = CUMULATIVE
Q Threshold = 20
Q FilterMaxScan = 1

[ApplyDirective]
ApplyFirst = ChromaCrawlReduction
MaxCycles = 1

; Define scaling parameters (required only if scaling images)
[ScaleParams]
proportional
x=320

; Output section (required)
[IO]

[Output]
OutputDir = c:\dibs

[Operations]
Scale
NTSC
DIB24

[Input]
c:\dibs\24bpp\refract.dib
```

Writing Filter Scripts

Filter scripts for Conver24 are ASCII text files. Each filter script consists of up to three sections: optional filter parameters and image scaling sections, and a required input/output section.

The Filter Parameters Section

The filter parameters section specifies the parameters of the filters used to process images. It is optional and not required if you are only resizing images or converting images to a different file format.

Note The numerous filter parameters provide maximum flexibility in processing images. However, to avoid being overwhelmed by the number of possible filter configurations, begin with the filter parameters provided in the SAMPLE.F24 filter script. These parameters provide a reasonable starting point for filtering images for NTSC display.

The filter parameters section must begin with the following header:

```
[NTSCParams]
```

This section consists of four parts, identified by the following headers:

Part	Header
Filter Definitions	[Filter1], [Filter2], ...
Chroma Crawl Reduction	[ChromaCrawlReduction]
Flicker Reduction	[FlickerReduction]
Apply Directive	[ApplyDirective]

The filter definitions part ([Filter1], ...) should always be at the beginning of the filtering-process section and the [ApplyDirective] part should always be at the end.

Filter-Definitions Part

In the filter-definitions part, you can define up to six different filters, each referenced by the index in the header of its definition section. Each definition consists of 3 parameters: xdim, ydim, and weights. The weights parameter lists the convolution weights, which determine the characteristics of a filter.

Chroma-Crawl Reduction Part

The chroma-crawl reduction part specifies horizontal filtering parameters for each of the color components, Y, I, and Q. For each component, there are four filtering parameters: FilterIndex, FilterOption, Threshold, and FilterMaxScan. A space must separate the component name (Y, I, or Q) and each parameter name on each line.

FilterIndex specifies which of the defined filters to use.

FilterOption specifies whether the filtering process is normal or cumulative. With normal filtering, unaltered values of surrounding pixels are used in calculating a filtered pixel value. With cumulative filtering, the altered values of surrounding pixels are used. For normal filtering set FilterOption to NORMAL; for cumulative filtering, set FilterOption to CUMULATIVE. For most images, cumulative filtering provides better results.

For each of the 3 color components, there is a Threshold parameter. This parameter must be set to an integer value. The range of the threshold is recommended to be -1 to 307, inclusive. A threshold of -1 or lower is equivalent to global, non-adaptive filtering, while a threshold of 307 or higher exceeds the maximum possible difference of any two Y, I, or Q values, and is thus equivalent to no filtering.

Filtering of a line of Y, I, or Q components is repeated until every pair of adjacent components differs by an amount no greater than the specified threshold. The parameter FilterMaxScan specifies a limit to the number of times the filter is repeated. A value of zero for FilterMaxScan is equivalent to no filtering as in the chroma-crawl reduction specifications for I and Q in the preceding example.

Flicker-Reduction Part

The flicker-reduction part is identical to the chroma-crawl reduction part, except the filtering is done along the vertical direction instead of the horizontal direction.

Apply Directive Part

The apply directive part specifies the number of times to apply the cycle of horizontal and vertical filtering, and which direction is to be applied first in each cycle.

The Image-Scaling Section

The optional image-scaling section specifies the dimensions for resizing images. It must begin with the following header:

```
[ScaleParams]
```

For proportional scaling, specify the proportional parameter and either the x or y parameter as follows:

```
[ScaleParams]
proportional
x = 160
```

For scaling to absolute dimensions, specify the x and y parameters as follows:

```
[ScaleParams]
x = 320
y = 200
```

The Input/Output Section

The input/output section specifies the operations to perform and the input and output filenames. This section consists of three parts identified by the following headers:

Part	Header
Input	[Input]
Output	[Output]
Operations	[Operations]

Input Part

The input part specifies a list of the files to process as follows:

```
[Input]
c:\dibs\24bpp\mickey.dib
c:\dibs\24bpp\goofy.dib
```

Acceptable input image formats include the following:

- 24-bit Windows DIB
- 24-bit OS/2 DIB
- 16-bit, 24-bit, and 32-bit TARGA

Wildcards are not allowed for input filenames.

Output Part

The output part uses the OutputDir parameter to specify where the output files are to be written:

```
[Output]
OutputDir = c:\dibs
```

Operations Part

The operations part specifies the operations to be performed on each of the images as well as the output format to use for the result. Use the NTSC parameter to specify that the image be filtered and the Scale parameter to specify that the image be scaled. Each of these parameters is optional–you can specify one or both. If you just want to convert an image to a different format, don't specify either of these parameters.

A third required parameter specifies the output format:

Parameter	Output Format
TGA24	24-bit Targa
DIB24	24-bit DIB
RGB555	16-bit RGB555 DIB
RGB565	16-bit RGB565 DIB
TYUV8	8-bit YUV DIB
TYUV16	16-bit YUV DIB

For example, the following input/output section instructs Conver24 to scale an image named REFRACT.DIB and write it as a RGB555 DIB to the C:\DIBS directory:

```
[IO]

[Output]
OutputDir = c:\dibs

[Operations]
Scale
RGB555

[Input]
refract.dib
```

The [IO], [Output], [Operations], and [Input] parts must be listed in the order shown in the example above.

APPENDIX A

VIS Memory-Cartridge Services

The Tandy Video Information System (VIS) uses a removable Save-It memory cartridge to provide applications with a small amount of non-volatile storage. Applications can use this storage for several purposes including saving user preferences or application-state information. This appendix describes the Modular Windows memory-cartridge services, provides a reference to the memory-cartridge API, and shows how applications can use this API to format, read, and write to memory cartridges.

About Memory Cartridges

Compared to magnetic media, memory cartridges contain a very small amount of memory (32K is a standard size). Memory cartridges can be either ROM (read-only), RAM (read/write), or a combination of ROM and RAM. Because the overhead of a FAT file system would consume too much of the memory on a memory cartridge, Modular Windows uses a binary file structure with one level of directory information.

Memory-Cartridge Function Overview

The following memory-cartridge functions allow applications to format, register, read, write, and retrieve the status of a memory cartridge:

mcAlloc
Allocates a given amount of memory on a memory cartridge.

mcInit
Initializes the MS-DOS® memory-cartridge library. This function is available only to MS-DOS applications—initialization occurs automatically with Windows applications.

mcRead
Reads a given block of memory from the currently registered section on a memory cartridge.

mcRegister
Registers a section on a memory cartridge as the current section for subsequent read and write operations.

mcStatus
Retrieves status information about a memory cartridge.

mcWrite
Writes a given block of memory to the currently registered section on a memory cartridge.

For detailed information about these functions, see "Memory-Cartridge Function Directory," later in this appendix.

Using the Memory-Cartridge Services

Information on a memory cartridge can only be accessed by using the memory-cartridge services discussed in this appendix. MS-DOS and Windows file I/O functions cannot be used to access a memory cartridge.

Registering Memory-Cartridge Sections

The information on a memory cartridge is organized into sections. Before reading, writing, or allocating memory on a memory cartridge, applications must register the section to be referenced by calling the **mcRegister** function. Read, write, and allocation requests are directed to the currently registered section. Only one section can be registered at a time—no handles or identifiers are required to reference a section. If **mcRegister** is called to register a section that doesn't already exist, it creates a new section using the given section name.

To create a new section on a memory cartridge, first register the section using the **mcRegister** function and then allocate space for the section using **mcAlloc**.

Naming Memory-Cartridge Sections

When an application registers a memory-cartridge section, it must provide a name for the section. Applications should adhere to the following guidelines when naming memory-cartridge sections:

- All characters are valid for section names.
- There is a limit of 64 characters for a section name (including the terminating NULL character).

- The percent sign (%) is a special character used by the memory cartridge management utility (MCMan). This character is used to separate the application name from the name of the section and does not appear as part of the section name presented by MCMan. Use the format as shown in the following example:

```
Application Name%sSection Name%
```

- Use descriptive names when naming memory-cartridge sections. The following are examples of descriptive section names:

```
Chess 3D%sHigh Scores%
Galactic Donuts%sLast Game%
```

See "Handling Full Memory Cartridges," later in this appendix, for details on using MCMan.

Handling Missing and Unformatted Cartridges

Because the memory cartridge is removable, it might not be present when applications need to read or write to it. Use the following guidelines to prompt users when a memory cartridge is not present or when one is present but not formatted:

- If the cartridge is not present, ask the user to insert it.
- If the cartridge is present but not formatted, format it. It's not necessary to query or notify the user before formatting the cartridge.

See the entry for the **mcRegister** function in the "Memory-Cartridge Function Directory," later in this appendix, for more information about formatting memory cartridges.

Handling Non-Existent Sections

If an application tries to register a section on a memory cartridge, there can be two reasons the section does not exist: either the wrong memory cartridge is inserted or the application has not yet written to the cartridge. There is no way for the application to distinguish between these two situations other than to query the user. There are two approaches for handling non-existent sections:

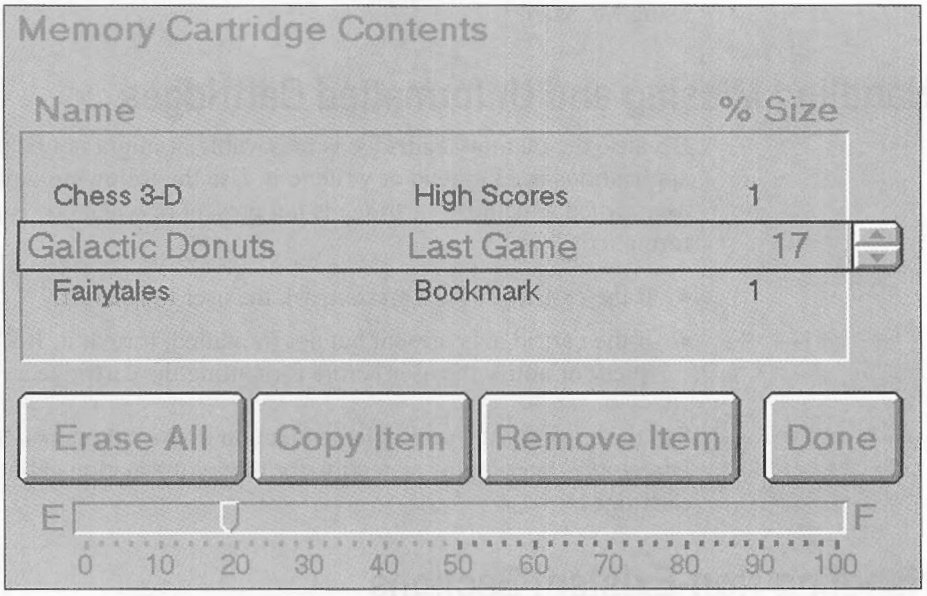
1. If the information on the memory cartridge is not critical to the operation of the application (such as high scores in a game), the application should just create the necessary sections on the memory cartridge, without querying the user.
2. If the information is critical to the operation of the application, it can request that the user insert the cartridge associated with the application.

Handling Full Memory Cartridges

If there is not enough free memory on a cartridge for an application to write to it, the application should attempt to free some space by deleting or reducing sections that are known to the application. If there is still not enough memory available on the cartridge, the application can call the MCMan memory cartridge management utility.

About the MCMan Utility

The MCMan utility allows users to delete sections on a memory cartridge when it becomes full. The following illustration shows the appearance of the main window of MCMan:



MCMan memory cartridge management utility for Tandy VIS systems

MCMan allows users to erase the entire cartridge, remove selected sections, and copy sections to other memory cartridges.

Running MCMan

To run MCMan, use the **WinExec** function, as shown in the following example:

```
WinExec("mcman", SW_SHOW);
```

MCMan accepts a single optional command-line argument to tell MCMan the amount of memory the application needs. The argument has the following syntax:

\m:*size*

The *size* parameter is a decimal number specifying the amount of memory in bytes that the application needs. MCMan uses this value to display a graph to aid the user in determining when enough memory has been freed. For example, the following statement runs MCMan to request that the user free 2K of memory:

```
WinExec("mcman \\m:2048", SW_SHOW);
```

Note MCMan is available in ROM for applications using the high-resolution display driver. The Modular Windows SDK also includes a dual-mode version of MCMan that runs in both low-resolution and high-resolution modes.

Using Memory-Cartridge Services with MS-DOS Applications

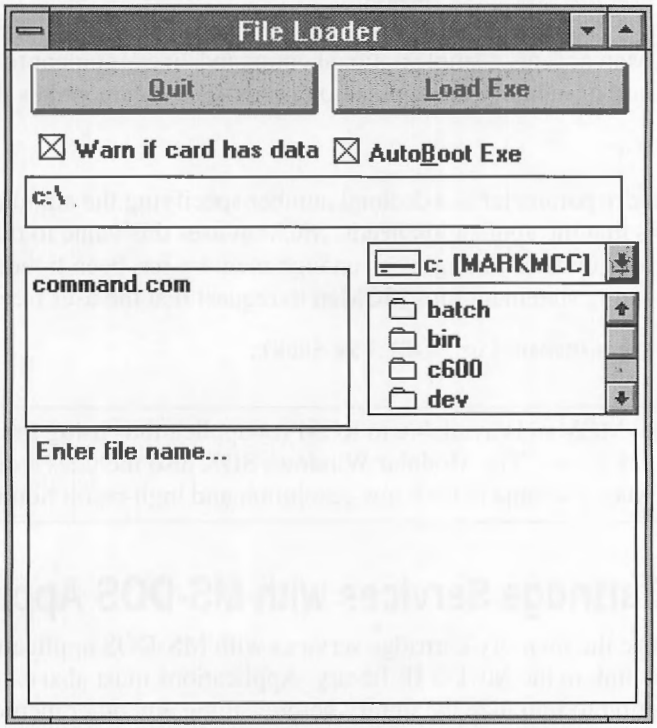
To use the memory-cartridge services with MS-DOS applications, applications must link to the MCD.LIB library. Applications must also call the **mcInit** function to initialize the library before calling any other memory-cartridge functions.

Installing Applications on Memory Cartridges

In addition to sections registered by applications, a memory cartridge can have one executable file that is loaded at startup. The MCLoadr utility is provided in the SDK to write files on a memory cartridge using a PC-based memory-cartridge unit. MCLoadr is a Windows 3.1 application (however, it is not designed to operate on VIS players).

Note Only one executable file can be resident on a memory cartridge. Applications cannot access dynamic-link libraries residing on a memory cartridge.

MCLoadr allows applications to load a single file (executable or data) onto a memory cartridge. The following illustration shows the main window of MCLoadr:



Main window of the MCLoadr utility

MCLoadr is automatically installed when you install the Modular Windows SDK. Its icon label in the Modular Windows SDK Program Manager group says "Memory Cartridge Utility."

Note Before you run MCLoadr, be sure to run GBIOS.COM. You must run GBIOS.COM before starting Windows.

▶ **To use the MCLoadr utility:**

1. Double-click the icon in the Modular Windows SDK.
2. Insert a memory cartridge into the cartridge slot. You can use a blank, unformatted cartridge or one that already contains information.
3. If you want the program to launch automatically when the cartridge is inserted in the user's system, select the Autoboot Exe check box.
4. Use the directory list box at the right of the dialog box to select the location of the file you want to copy to the memory cartridge.
5. Use the file list box at the left of the dialog box to select the file you want to copy.
6. Choose the Load Exe button. As the system copies the file to the memory cartridge, it displays a list of messages.

   If the memory cartridge already contains information or has previously been formatted, the system asks whether you want to overwrite the information on the cartridge. If so, choose the OK button. If not, choose the Cancel button.

   When you see the message "Done," the copy process is complete.
7. After the copy operation is complete, you can copy a file to another cartridge, or exit the utility.
8. To exit, choose the Quit button.

Memory-Cartridge Function Directory

The following is an alphabetical list of memory cartridge functions. Each entry contains the following items:

- The type, name, and description of input parameters
- The syntax for the function
- The purpose of the function
- A description of the return value
- Optional comments about using the function
- Optional examples showing how to use the function
- Optional cross references to other functions, macros, messages, and data structures

mcAlloc

Syntax

DWORD mcAlloc(*dwSize*)

The **mcAlloc** function allocates the given amount of memory to the currently registered section on a memory cartridge. If memory is already allocated to the currently registered section, it is reallocated to the given size. If the given size is zero, the section is deleted from the cartridge, and the application will no longer be registered to the deleted section.

Parameters

DWORD *dwSize*
Specifies the amount of memory to allocate.

Return Value

The return value is zero if the function is successful; otherwise, it is one of the following error codes:

MCERR_BADCARD
The cartridge is corrupt.

MCERR_CARDCHANGED
The cartridge that is present is different than the cartridge that was present on the last call to **mcRegister**.

MCERR_READONLY
The current memory cartridge is a read-only cartridge.

MCERR_NOTREGISTERED
There is no currently registered section.

MCERR_NOCARD
There is no memory cartridge present.

MCERR_CARDFULL
There is not enough memory available on the memory cartridge.

Comments

You must call **mcRegister** to register a section before allocating memory. The section is not actually created until **mcAlloc** is called with a non-zero value for the *dwSize* parameter. If the amount of memory requested is smaller than the amount currently allocated, memory is deallocated from the end (higher address) of the currently allocated space. If the amount of memory requested is larger than the amount currently allocated, additional memory is allocated at the end of the currently allocated space. The content of the previously allocated memory is preserved.

See Also

mcRegister

mcInit

Syntax

DWORD mcInit(*wFlags*)

The **mcInit** function initializes the memory-cartridge library.

Parameters

WORD *wFlags*
Specifies one of the following flags indicating whether the application is initializing or terminating its use of the library:

MCINIT_START
Initialize the services of the memory-cartridge library.

MCINIT_END
Terminate the services of the memory-cartridge library.

Return Value

The return value is zero if the function is successful; otherwise, it is the following error value:

MCERR_NOCORE
There is not enough memory available in the system.

Comments

This function is available only to applications using the MS-DOS version of the memory-cartridge library. MS-DOS applications must initialize the memory-cartridge library before using any memory-cartridge functions and terminate the services of the library before exiting.

mcRead

Syntax

DWORD mcRead(*dwStart*, *dwLen*, *hpBuf*)

The **mcRead** function reads the given number of bytes from the currently registered section on a memory cartridge into a given buffer.

Parameters

DWORD *dwStart*
Specifies the location to begin reading as an offset into the currently registered section.

DWORD *dwLen*
Specifies the number of bytes to read.

HPBYTE *hpBuf*
Specifies a huge pointer to a buffer of at least *dwLen* bytes.

Return Value

The return value is zero if the function is successful; otherwise, it is one of the following error values:

MCERR_NOTREGISTERED
There is no currently registered section.

MCERR_CARDCHANGED
The cartridge that is present is different than the cartridge that was present on the last call to **mcRegister**.

MCERR_NOCARD
There is no memory cartridge present.

MCERR_OUTOFRANGE
Attempted to read past end of section.

MCERR_BADCARD
The file system on the cartridge has been corrupted.

MCERR_BADDATA
The data in the currently registered section has been corrupted.

See Also **mcAlloc**, **mcRegister**, **mcWrite**

mcRegister

Syntax **DWORD mcRegister**(*lpszSection*, *wOptions*)

The **mcRegister** function registers a section on a memory cartridge as the current section for subsequent read and write operations. If the given section name does not exist, the new section is created when space is allocated for the new section using **mcAlloc**.

Parameters

LPSTR *lpszSection*
Specifies the name of the section to register.

WORD *wOptions*
Specifies one option for registering the section:

MC_AUTOFORMAT
Formats unformatted or corrupted memory cartridges before registering the section.

Return Value The return value is zero if the function is successful; otherwise, it is one of the following error values:

MCERR_BADSECTIONNAME
The given section name is not valid.

MCERR_NOCARD
There is no memory cartridge present.

MCERR_CARDUNFORMATTED
The cartridge is not formatted and the MC_AUTOFORMAT flag was not specified.

MCERR_UNSUPPORTEDCARDVERSION
The cartridge is recognized but not supported.

MCERR_BADCARD
The cartridge is corrupt.

MCERR_NOCORE
A request to allocate system memory failed.

MCERR_CARDFULL
An attempt was made to register a new section and the cartridge is full.

Comments

If the registration is not successful, there is no currently registered section and calls to other memory-cartridge functions will fail. New sections are not created until the next **mcAlloc** call. To delete a section from a memory cartridge, register the section using **mcRegister** and then call **mcAlloc** with a size of zero. To unregister the currently registered section, call **mcRegister** with *lpszSection* set to NULL. Applications are not required to unregister sections, but doing so prevents other tasks from accessing the section.

See Also

mcAlloc, mcStatus, mcRead, mcWrite

mcStatus

Syntax

DWORD mcStatus(*lpMCS*)

The **mcStatus** function retrieves information about a memory cartridge.

Parameters

LPMCSTATUS *lpMCS*
Specifies a far pointer to an **MCSTATUS** structure. Set the **wHdrSize** field to the size of the **MCSTATUS** structure before calling **mcStatus**.

Return Value

The return value is zero if the function is successful; otherwise, it is one of the following error values:

MCERR_CARDUNFORMATTED
The memory cartridge is not formatted.

MCERR_BADCARD
The file system on the cartridge has been corrupted.

MCERR_BADPARAMETER
The *lpMCS* pointer is NULL.

MCERR_NOCARD
There is no memory cartridge present.

MCERR_UNSUPPORTEDCARDVERSION
The cartridge is recognized but not supported.

Comments

Applications don't need to register a section before calling **mcStatus**. If there is no section registered, the **dwUsedRAM** and the **mcidReg** fields will be zero. The amount of memory reported in the **dwUsedRAM** field might be slightly larger than the amount requested by **mcAlloc**.

See Also

mcRegister, **MCSTATUS**

mcWrite

Syntax

DWORD mcWrite(*dwStart*, *dwLen*, *hpBuf*)

The **mcWrite** function writes the given number of bytes to the currently registered section on a memory cartridge.

Parameters

DWORD *dwStart*
Specifies the location to begin writing as an offset into the currently registered section.

DWORD *dwLen*
Specifies the number of bytes to write.

HPBYTE *hpBuf*
Specifies a huge pointer to a buffer of at least *dwLen* bytes.

Return Value

The return value is zero if the function is successful; otherwise, it is one of the following error values:

MCERR_NOTREGISTERED
There is no currently registered section.

MCERR_NOCARD
There is no memory cartridge present.

MCERR_OUTOFRANGE
Attempted to write past end of section.

MCERR_READONLY
The current memory cartridge is a read-only cartridge.

MCERR_BADCARD
The file system on the cartridge has been corrupted.

MCERR_BADDATA
The data in the currently-registered section has been corrupted.

MCERR_CARDCHANGED
The cartridge that is present is different than the cartridge that was present on the last call to **mcRegister**.

See Also **mcAlloc, mcRegister, mcRead**

Memory-Cartridge Data-Structure Directory

The following is an alphabetical list of memory-card data structures. Each entry gives the structure definition along with the type and a description of the contents of each of the fields in the structure.

MCSTATUS

The **MCSTATUS** structure contains information about a memory cartridge.

```
typedef struct {
  WORD wHdrSize;
  WORD wStatus;
  DWORD dwTotalROM;
  DWORD dwTotalRAM;
  DWORD dwFreeRAM;
  DWORD dwUsedRAM;
  MCID mcid;
  MCID mcidReg;
  WORD wCardVersion;
  BYTE abReserved[8];
}MCSTATUS;
```

Fields

wHdrSize
Specifies the size of the **MCSTATUS** structure.

wStatus
Specifies one or more of the following flags indicating the status of the memory cartridge:

MC_CARDPRESENT
A memory cartridge is present.

MC_CARDCHANGED
A new memory cartridge has been inserted since the last **mcRegister** call.

dwTotalROM
Specifies the total amount of ROM on the memory cartridge.

dwTotalRAM
Specifies the total amount of RAM on the memory cartridge.

dwFreeRAM
Specifies the amount of free RAM on the memory cartridge.

dwUsedRAM
Specifies the amount of RAM used by the currently registered section. The amount of memory reported might be slightly larger than the amount requested by **mcAlloc**.

mcid
Specifies the memory-cartridge ID of the cartridge.

mcidReg
Specifies the memory-cartridge ID of the cartridge present on the last successful **mcRegister** call.

wCardVersion
Specifies the format and capabilities of the cartridge. Currently, all memory cartridges are version 0x0001.

abReserved[8]
Reserved for future use.

See Also

mcStatus

APPENDIX B

VIS Programming Notes

This appendix contains information specific to programming applications for VIS systems. It covers the following topics:

- Detecting if an application is running on a VIS player
- Exiting application and ejecting disc from CD-ROM drive
- Setting mixer levels
- Authoring MIDI files
- Authoring video files
- YUV file formats

Much of the sample code in this appendix is taken from the WinShell sample application.

Developing Applications for VIS

To develop applications for VIS, you must obtain additional tools from the VIS manufacturer. For information about obtaining these tools, contact the following:

Dennis Tanner
Director of Strategic Software Marketing
Tandy Corporation
916 One Tandy Center
Ft. Worth, TX 76102
(817) 390-3477
(817) 878-6669 (FAX)

Detecting if an Application is Running on a VIS Player

Many of the examples in this appendix use function calls that are unique to the VIS system. Some of these calls might fail on other TV-based players or on PCs. Applications should detect what type of hardware they are running on before making hardware-dependent function calls. The following code fragment is a function that can be used to detect if an application is running on a VIS system:

```
/*
 * DetectVIS
 *
 * Determines if host is a VIS player by looking for the presence of
 *   the VIS launcher
 *
 * Returns TRUE if the launcher exists; FALSE if not.
 */
#define TANDY_2F_HOOK   0x0081
#define  OEM_PRESENCE   0x0000
DetectVIS(void)
{
    BOOL    bReturn=FALSE;

    _asm
    {
        /* Check for VIS launch module */
        mov     ah, TANDY_2F_HOOK
        mov     al, OEM_PRESENCE
        int     2Fh

        /* If the launcher exists, it will return 0xFF */
        cmp     al, 0ffh
        jne     NoVIS

        mov     bReturn, TRUE

        NoVIS:
    }

    return bReturn;
}
```

Exiting an Application and Ejecting the CD-ROM Disc

The following example is a function that can be used to exit an application and eject the disc from the CD-ROM drive:

```
/*
 * CleanVISExit
 *
 * Ejects the disc from the CD-ROM and returns control to the
 *    launch module
 *
 * Returns: You will not return from this function.
 *
 */
#define TANDY_2F_HOOK           0x0081
#define OEM_DOOR_OPEN_ACTION    0x0011
void CleanVISExit(void)
{
    _asm
    {
        /* Open the door */
        mov     ah, TANDY_2F_HOOK
        mov     al, OEM_DOOR_OPEN_ACTION
        int     2Fh

        /* Goodbye */
        int     19h
    }
}
```

Setting Mixer Levels

VIS players have an audio mixer for the compact-disc, waveform, and MIDI synthesizer audio signals. The default settings for the mixer are too low to be useful—if your application uses audio, you should increase the mixer settings corresponding to the type(s) of audio the application uses to provide a comfortable listening level for users. To determine the right mixer-level settings, test the application in a situation that allows the television to be switched between broadcast reception and VIS, and test the application in comparison with other VIS applications.

The WinShell sample application provides routines you can use to set VIS mixer levels.

Authoring MIDI Files

The MIDI synthesizer in VIS systems can operate in either of the following two modes:

- General MIDI mode
- Microsoft base-level mode

In Microsoft base-level mode, the synthesizer responds to MIDI channels 13 through 16, with percussion instruments on channel 16. In general MIDI mode, the synthesizer responds to MIDI channels 1 through 16, with percussion instruments on channel 10. For details on base-level synthesizers, see the *Multimedia Programmer's Guide* in the Microsoft Windows 3.1 SDK. For details on general MIDI synthesizers, see the General MIDI Mode specification. This specification is available from the International MIDI Association (IMA) at the following address:

International MIDI Association
5316 West 57th Street
Los Angeles, CA 90056

Applications can play MIDI files authored to use either base-level mode or general MIDI mode. The default mode for the VIS MIDI synthesizer is base-level mode—if applications use MIDI files authored for general MIDI mode, they must set the synthesizer to general MIDI mode.

Setting General MIDI Mode

To set the synthesizer to general MIDI mode, applications must send the synthesizer a GENERAL MIDI MODE ON message. This message is defined in the General MIDI Mode specification as the following sequence of hexadecimal bytes:

```
F0 7E 7F 09 01 F7

   F0 7E       = Universal Non-Real Time SysEx header
   7F          = Device ID (broadcast)
   09          = Sub-ID #1, General MIDI Message
   01          = Sub-ID #2, General MIDI On
   F7          = EOX
```

The easiest way to send the synthesizer a GENERAL MIDI MODE ON message is to embed the message in a MIDI file such that it is the first message sent to the synthesizer (before any note, program change, or volume-control messages). Applications can also use the **midiOutLongMsg** function to send the message to the synthesizer.

Setting Microsoft Base-Level Mode

To set the synthesizer to Microsoft base-level mode, applications must send the synthesizer a GENERAL MIDI MODE OFF message. This message is defined in the General MIDI Mode specification as the following sequence of hexadecimal bytes:

```
F0 7E 7F 09 02 F7

   F0 7E       = Universal Non-Real Time SysEx header
   7F          = Device ID (broadcast)
   09          = Sub-ID #1, General MIDI Message
   02          = Sub-ID #2, General MIDI Off
   F7          = EOX
```

The easiest way to send the synthesizer a GENERAL MIDI MODE OFF message is to embed the message in a MIDI file such that it is the first message sent to the synthesizer (before any note, program change, or volume control messages). Applications can also use the **midiOutLongMsg** function to send the message to the synthesizer.

Authoring Video Files

Use the following guidelines when authoring AVI files for VIS systems:

- Compress the files using the Microsoft RLE compressor. Modular Windows for VIS does not support playback of AVI files compressed using the Microsoft Video 1 compressor.
- Compress the files for CD-ROM (150K/sec) target with frames padded for CD-ROM playback.
- Ensure the files are not fragmented when placed on CD-ROM.
- To improve performance, try using lower bandwidth audio, such as 8-bit 11kHz monophonic.

YUV DIB Formats

VIS players support 8- and 16-bit YUV video modes. Modular Windows does not provide a display driver for these video modes, but does provide support for displaying YUV images with the **DisplayDib** function. This section describes the YUV file formats supported by Modular Windows on VIS players. The Convert24 utility will convert 24- and 32-bit Targa (TGA) files to these YUV file formats.

Note The YUV formats provide higher quality images on televisions than the palettized RGB formats.

Pixel information in YUV images is encoded into luminance and chrominance components. Luminance is the brightness of the image and is notated as the Y element in YUV images. Chrominance is the color information and is notated as the U (blue-green) and V (red-green) elements in YUV images. To take advantage of the lower sensitivity of the human eye to color as compared to brightness, the YUV DIB formats encode the Y information at high resolution and compress the U and V information.

There are two YUV DIB formats used by VIS players:

- TYUV8—uses five bits of Y for each pixel, with five bits of U and five bits of V shared between each group of four pixels. The 5-bit values are compressed from 8-bit values by using a companding table.
- TYUV16—uses eight bits of Y for each pixel, with eight bits of U and eight bits of V shared between each group of two pixels.

These YUV formats use compression and differential encoding techniques based on Tandy video hardware, which use the "TYUV" designation to distinguish them from other more generic YUV formats.

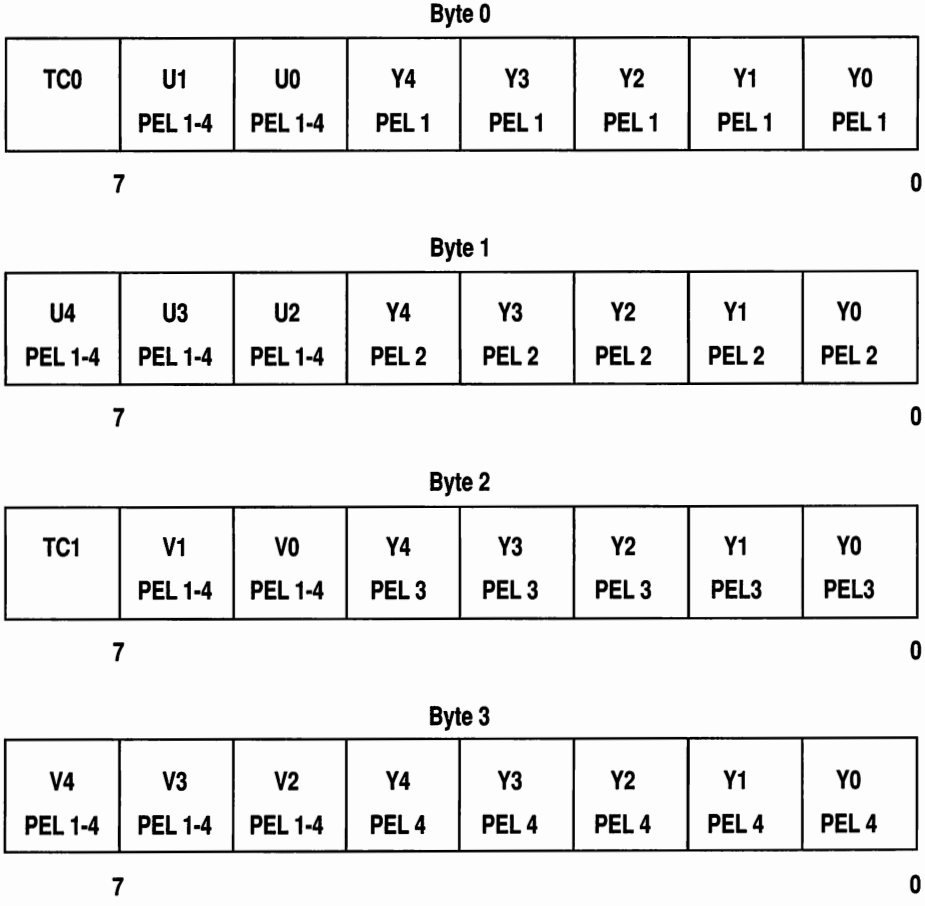
BITMAPINFOHEADER Structure for TYUV8 and TYUV16 DIBs

The following table contains information on the fields of the **BITMAPINFOHEADER** structure for TYUV DIBs:

Field	Description
biSize	Size in bytes of the **BITMAPINFOHEADER** structure.
biWidth	Width of the bitmap in pixels.
biHeight	Height of the bitmap in pixels.
biPlanes	Set to 1.
biBitCount	Set to 8 for TYUV8; 16 for TYUV16.
biCompression	Set to the four-character code 'TYUV'.
biSizeImage	Size in bytes of the image.
biXPelsPerMeter	Horizontal resolution in pixels per meter.
biYPelsPerMeter	Vertical resolution in pixels per meter.
biClrUsed	Set to 0.
biClrImportant	Set to 0.

TYUV8 Pixel Encoding

The TYUV8 DIB format uses 32 bits to encode the pixel information for each group of 4 pixels. There are 5 bits of Y for each pixel along with 5 bits of U and 5 bits of V shared between each group of 4 pixels. The following illustration shows the pixel encoding for a group of four pixels in TYUV8 format:



Pixel encoding for TYUV8 DIB

Companding of Luminance and Chrominance Values

Luminance (Y) and chrominance (U and V) values are compressed from 8-bit values to 5-bit values using the following companding table:

Original 8-Bit Value		Compressed 5-Bit Value	Expanded 5-Bit Value
0		0	0
1	(-255)	1	1
2	(-254)	2	2
3	(-253)	3	3
4	(-252)	4	4
5	(-251)	5	5
6 to 7	(-250 to -249)	6	6
8 to 10	(-248 to -246)	7	9
11 to 14	(-245 to -242)	8	12
15 to 19	(-241 to -237)	9	17
20 to 25	(-236 to -231)	10	22
26 to 33	(-230 to -223)	11	29
34 to 43	(-222 to -213)	12	38
44 to 56	(-212 to -200)	13	50
57 to 76	(-199 to -180)	14	66
77 to 106	(-179 to -150)	15	91
107 to 149	(-149 to -107)	16	128
150 to 179	(-106 to -77)	17	165
180 to 199	(-76 to -57)	18	190
200 to 212	(-56 to -44)	19	206
213 to 222	(-43 to -34)	20	218
223 to 230	(-33 to -26)	21	227
231 to 236	(-25 to -20)	22	234
237 to 241	(-19 to -15)	23	239
242 to 245	(-14 to -11)	24	244
246 to 248	(-10 to -8)	25	247
249 to 250	(-7 to -6)	26	250

Continued

Original 8-Bit Value		Compressed 5-Bit Value	Expanded 5-Bit Value
251	(-5)	27	251
252	(-4)	28	252
253	(-3)	29	253
254	(-2)	30	254
255	(-1)	31	255

TYUV8 Luminance Encoding

The luminance (Y) information is differentially encoded—each 5-bit value (after expansion to an 8-bit value) represents the difference in luminance from the previous pixel. For the first pixel of each scan line, the 5-bit Y value represents the upper 5 bits of an 8-bit initialization value for a running sum (the lower 3 bits are zero). Each subsequent 5-bit value is expanded to 8 bits and added to the running sum to generate the next 8-bit Y value.

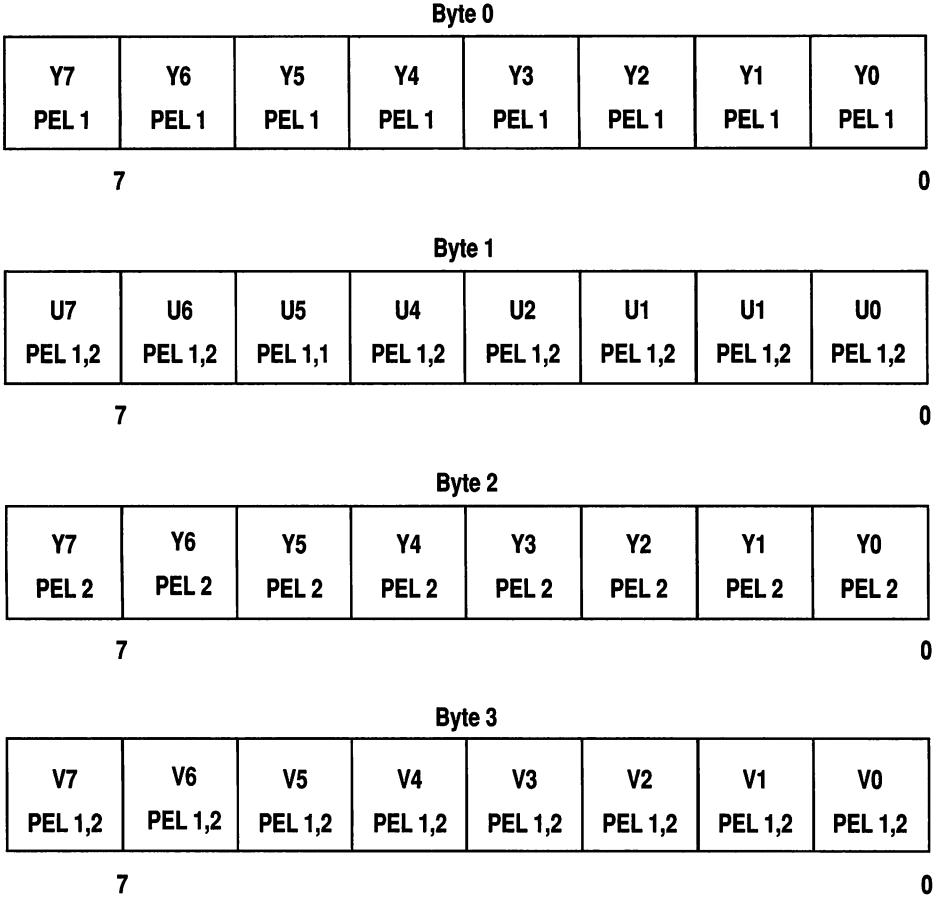
TYUV8 Chrominance Encoding

The chrominance (U and V) information is specified as a single 5-bit companded value for each group of four pixels, representing the average color for those pixels. In addition, two bits (TC0 and TC1) supply a transition code specifying where the transition from the present U and V values to the next values should take place. This method allows the encoding software to absorb the burden of determining the locations of significant color transitions. The following table shows where the transitions occur for the different transition codes. In the table, "A" represents U and V values for a given group of four pixels and "B" represents U and V values for the next group of four pixels.

Pixel 1	Pixel 2	Pixel 3	Pixel 4
Transition Code: 00			
(A + B)/2	B	B	B
Transition Code: 01			
A	(A + B)/2	B	B
Transition Code: 10			
A	A	(A + B)/2	B
Transition Code: 11			
A	A	A	(A + B)/2

TYUV16 Pixel Encoding

The TYUV16 DIB format uses 32 bits to encode the pixel information for each group of two pixels. There are 8 bits of Y for each pixel along with 8 bits of U and 8 bits of V shared between each pair of pixels. The following illustration shows the pixel encoding for a pair of pixels in YUV16 format:



Pixel encoding for TYUV16 DIB

TYUV16 Chrominance Encoding

The chrominance (U and V) information is interpolated for the second pixel in each pair of pixels. The chrominance value for these pixels is equal to the average of the chrominance of the previous and the next pixels. The following table illustrates this averaging technique. In the table, "A" represents U and V values for a given pair of pixels, and "B" represents U and V values for the next pair of pixels.

Pixel 1	Pixel 2	Pixel 3	Pixel 4
A	(A+B)/2	B	(B+C)/2

Index

See also Microsoft Modular Windows Design Guide Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®