Multimedia Programmer's Reference

Microsoft® WINDOWS™
SOFTWARE DEVELOPMENT KIT

Document No. PC30211-0492

Document Conventions

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

Convention	Meaning
Bold text	Denotes a term or character to be typed literally, such as a resource-definition statement or function name (**MENU** or **CreateWindow**), an MS-DOS command, or a command-line option (**/nod**). 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.
[ ]	Enclose optional parameters.
|	Separates an either/or choice.
...	Specifies that the preceding item may be repeated.

`BEGIN`
.
.
.
`END`
	Represents an omitted portion of a sample application.

In addition, certain text conventions are used to help you understand this material:

Convention	Meaning
SMALL CAPITALS	Indicate the names of keys, key sequences, and key combinations—for example, ALT+SPACEBAR.
FULL CAPITALS	Indicate filenames and paths, most type and structure names (which are also bold), and constants.
`monospace`	Sets off code examples and shows syntax spacing.

color format and the width, in pixels, of the bitmap. If necessary, a scan line must be zero-padded to end on a 32-bit boundary. However, segment boundaries can appear anywhere in the bitmap. The scan lines in the bitmap are stored from bottom up. This means that the first byte in the array represents the pixels in the lower-left corner of the bitmap and the last byte represents the pixels in the upper-right corner.

The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. These members can have any of the following values:

Value	Meaning
1	Bitmap is monochrome and the color table contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the color table. If the bit is set, the pixel has the color of the second entry in the table.
4	Bitmap has a maximum of 16 colors. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	Bitmap has a maximum of 256 colors. Each pixel in the bitmap is represented by a 1-byte index into the color table. For example, if the first byte in the bitmap is 0x1F, the first pixel has the color of the thirty-second table entry.
24	Bitmap has a maximum of 2^{24} colors. The **bmiColors** (or **bmciColors**) member is NULL, and each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, for a pixel.

The **biClrUsed** member of the **BITMAPINFOHEADER** structure specifies the number of color indexes in the color table actually used by the bitmap. If the **biClrUsed** member is set to zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member.

An alternative form of bitmap file uses the **BITMAPCOREINFO**, **BITMAPCOREHEADER**, and **RGBTRIPLE** structures.

For a full description of the bitmap structures, see the *Microsoft Windows Programmer's Reference, Volume 3*.

1.1.2 Bitmap Compression

Windows versions 3.0 and later support run-length encoded (RLE) formats for compressing bitmaps that use 4 bits per pixel and 8 bits per pixel. Compression reduces the disk and memory storage required for a bitmap.

1.1.2.2 Compression of 4-Bits-per-Pixel Bitmaps

When the **biCompression** member of the **BITMAPINFOHEADER** structure is set to BI_RLE4, the DIB is compressed using a run-length encoded format for a 16-color bitmap. This format uses two modes: encoded mode and absolute mode.

Encoded Mode A unit of information in encoded mode consists of two bytes. The first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte.

The second byte contains two color indexes, one in its high-order nibble (that is, its low-order 4 bits) and one in its low-order nibble. The first pixel is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

The first byte of the pair can be set to zero to indicate an escape that denotes the end of a line, the end of the bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair. In encoded mode, the second byte has a value in the range 0x00 through 0x02. The meaning of these values is the same as for a DIB with 8 bits per pixel.

Absolute Mode In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. Each run must be aligned on a word boundary.

Following is an example of a 4-bit RLE bitmap (the one-digit hexadecimal values in the second column represent a color index for a single pixel):

Compressed data	Expanded data
03 04	0 4 0
05 06	0 6 0 6 0
00 06 45 56 67 00	4 5 5 6 6 7
04 78	7 8 7 8
00 02 05 01	Move 5 right and 1 down
04 78	7 8 7 8
00 00	End of line
09 1E	1 E 1 E 1 E 1 E 1
00 01	End of RLE bitmap

1.2 Icon-Resource File Format

An icon-resource file contains image data for icons used by Windows applications. The file consists of an icon directory identifying the number and types of icon images in the file, plus one or more icon images. The default filename extension for an icon-resource file is .ICO.

1.2.1 Icon Directory

Each icon-resource file starts with an icon directory. The icon directory, defined as an **ICONDIR** structure, specifies the number of icons in the resource and the dimensions and color format of each icon image. The **ICONDIR** structure has the following form:

```
typedef struct ICONDIR {
    WORD          idReserved;
    WORD          idType;
    WORD          idCount;
    ICONDIRENTRY  idEntries[1];
} ICONHEADER;
```

Following are the members in the **ICONDIR** structure:

idReserved
Reserved; must be zero.

idType
Specifies the resource type. This member is set to 1.

idCount
Specifies the number of entries in the directory.

idEntries
Specifies an array of **ICONDIRENTRY** structures containing information about individual icons. The **idCount** member specifies the number of structures in the array.

The **ICONDIRENTRY** structure specifies the dimensions and color format for an icon. The structure has the following form:

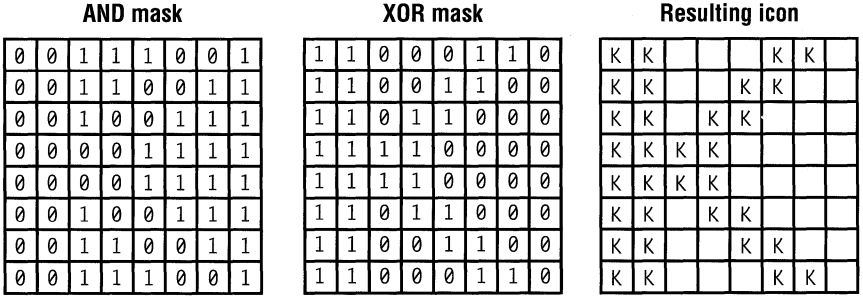
```
struct IconDirectoryEntry {
    BYTE  bWidth;
    BYTE  bHeight;
    BYTE  bColorCount;
    BYTE  bReserved;
    WORD  wPlanes;
    WORD  wBitCount;
    DWORD dwBytesInRes;
    DWORD dwImageOffset;
};
```

The XOR mask, immediately following the color table, is an array of **BYTE** values representing consecutive rows of a bitmap. The bitmap defines the basic shape and color of the icon image. As with the bitmap bits in a bitmap file, the bitmap data in an icon-resource file is organized in scan lines, with each byte representing one or more pixels, as defined by the color format. For more information about these bitmap bits, see Section 1.1, "Bitmap-File Formats."

The AND mask, immediately following the XOR mask, is an array of **BYTE** values, representing a monochrome bitmap with the same width and height as the XOR mask. The array is organized in scan lines, with each byte representing 8 pixels.

When Windows draws an icon, it uses the AND and XOR masks to combine the icon image with the pixels already on the display surface. Windows first applies the AND mask by using a bitwise AND operation; this preserves or removes existing pixel color. Windows then applies the XOR mask by using a bitwise XOR operation. This sets the final color for each pixel.

The following illustration shows the XOR and AND masks that create a monochrome icon (measuring 8 pixels by 8 pixels) in the form of an uppercase K:



1.2.3 Windows Icon Selection

Windows detects the resolution of the current display and matches it against the width and height specified for each version of the icon image. If Windows determines that there is an exact match between an icon image and the current device, it uses the matching image. Otherwise, it selects the closest match and stretches the image to the proper size.

If an icon-resource file contains more than one image for a particular resolution, Windows uses the icon image that most closely matches the color capabilities of the current display. If no image matches the device capabilities exactly, Windows selects the image that has the greatest number of colors without exceeding the number of display colors. If all images exceed the color capabilities of the current display, Windows uses the icon image with the least number of colors.

Following are the members in the **CURSORDIRENTRY** structure:

bWidth
Specifies the width of the cursor, in pixels.

bHeight
Specifies the height of the cursor, in pixels.

bColorCount
Reserved; must be zero.

bReserved
Reserved; must be zero.

wXHotspot
Specifies the x-coordinate, in pixels, of the hot spot.

wYHotspot
Specifies the y-coordinate, in pixels, of the hot spot.

lBytesInRes
Specifies the size of the resource, in bytes.

dwImageOffset
Specifies the offset, in bytes, from the start of the file to the cursor image.

1.3.2 Cursor Image

Each cursor-resource file contains one cursor image for each image identified in the cursor directory. A cursor image consists of a cursor-image header, a color table, an XOR mask, and an AND mask. The cursor image has the following form:

```
BITMAPINFOHEADER    crHeader;
RGBQUAD             crColors[];
BYTE                crXOR[];
BYTE                crAND[];
```

The cursor hot spot is a single pixel in the cursor bitmap that Windows uses to track the cursor. The **crXHotspot** and **crYHotspot** members specify the x- and y-coordinates of the cursor hot spot. These coordinates are 16-bit integers.

The cursor-image header, defined as a **BITMAPINFOHEADER** structure, specifies the dimensions and color format of the cursor bitmap. Only the **biSize** through **biBitCount** members and the **biSizeImage** member are used. The **biHeight** member specifies the combined height of the XOR and AND masks for the cursor. This value is twice the height of the XOR mask. The **biPlanes** and **biBitCount** members must be 1. All other members (such as **biCompression** and **biClrImportant**) must be set to zero.

1.3.3 Windows Cursor Selection

If a cursor-resource file contains more than one cursor image, Windows determines the best match for a particular display by examining the width and height of the cursor images.

Following are the members in the metafile header:

mtType
Specifies whether the metafile is stored in memory or recorded in a file. This member has one of the following values:

Value	Meaning
0	Metafile is in memory.
1	Metafile is in a file.

mtHeaderSize
Specifies the size, in words, of the metafile header.

mtVersion
Specifies the Windows version number. The version number for Windows version 3.0 and later is 0x300.

mtSize
Specifies the size, in words, of the file.

mtNoObjects
Specifies the maximum number of objects that can exist in the metafile at the same time.

mtMaxRecord
Specifies the size, in words, of the largest record in the metafile.

mtNoParameters
Not used.

3.2 Typical Metafile Record

The graphics device interface stores most of the GDI functions that an application can use to create metafiles in typical records.

A typical metafile record has the following form:

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Following are the members in a typical metafile record:

rdSize
Specifies the size, in words, of the record.

GDI function	Value
SetTextAlign	0x012E
SetTextCharExtra	0x0108
SetTextColor	0x0209
SetTextJustification	0x020A
SetViewportExt	0x020E
SetViewportOrg	0x020D
SetWindowExt	0x020C
SetWindowOrg	0x020B

For more information on GDI functions, see the *Microsoft Windows Programmer's Reference, Volume 2*. For more information on the function-specific metafile records, see Section 3.6, "Function-Specific Metafile Records."

3.3 Placeable Windows Metafiles

A placeable Windows metafile is a standard Windows metafile that has an additional 22-byte header. The header contains information about the aspect ratio and original size of the metafile, permitting applications to display the metafile in its intended form.

The header for a placeable Windows metafile has the following form:

```
typedef struct {
    DWORD   key;
    HANDLE  hmf;
    RECT    bbox;
    WORD    inch;
    DWORD   reserved;
    WORD    checksum;
} METAFILEHEADER;
```

Following are the members of a placeable metafile header:

key
Specifies the binary key that uniquely identifies this file type. This member must be set to 0x9AC6CDD7L.

hmf
Unused; must be zero.

bbox
Specifies the coordinates of the smallest rectangle that encloses the picture. The coordinates are in metafile units as defined by the **inch** member.

3.5 Sample of Metafile Program Output

This section describes a sample program and the metafile that it creates. The sample program creates a small metafile that draws a purple rectangle with a green border and writes the words "Hello People" in the rectangle.

```
MakeAMetaFile(hDC)
HDC hDC;
{
    HPEN      hMetaGreenPen;
    HBRUSH    hMetaVioletBrush;
    HDC       hDCMeta;
    HANDLE    hMeta;

    /* Create the metafile with output going to the disk. */

    hDCMeta = CreateMetaFile( (LPSTR) "sample.met");

    hMetaGreenPen = CreatePen(0, 0, (DWORD) 0x0000FF00);
    SelectObject(hDCMeta, hMetaGreenPen);

    hMetaVioletBrush = CreateSolidBrush((DWORD) 0x00FF00FF);
    SelectObject(hDCMeta, hMetaVioletBrush);

    Rectangle(hDCMeta, 0, 0, 150, 70);

    TextOut(hDCMeta, 10, 10, (LPSTR) "Hello People", 12);

    /* We are done with the metafile. */

    hMeta = CloseMetaFile(hDCMeta);

    /* Play the metafile that we just created. */

    PlayMetaFile(hDC, hMeta);
}
```

The resulting metafile, SAMPLE.MET, consists of a metafile header and six records. It has the following binary form:

```
0001          mtType... disk metafile
0009          mtSize...
0300          mtVersion
0000 0036     mtSize
0002          mtNoObjects
0000 000C     mtMaxRecord
0000          mtNoParameters
```

AnimatePalette

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0436.

rdParm
Contains the following elements:

Element	Description
start	First entry to be animated
numentries	Number of entries to be animated
entries	**PALETTEENTRY** blocks (for a description of the **PALETTEENTRY** structure, see the *Microsoft Windows Programmer's Reference, Volume 3*).

BitBlt

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

The **BitBlt** record stored by Windows versions earlier than 3.0 contains a device-dependent bitmap that may not be suitable for playback on all devices.

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0922.

rdParm
Contains the following elements:

Element	Description
DXE	Destination x-extent
DY	Y-coordinate of the destination origin
DX	X-coordinate of the destination origin
BitmapInfo	**BITMAPINFO** structure (for a description of the **BITMAPINFO** structure, see the *Microsoft Windows Programmer's Reference, Volume 3*).
bits	Actual device-independent bitmap bits

CreateBrushIndirect

```
struct {
    DWORD    rdSize;
    WORD     rdFunction;
    LOGBRUSH rdParm;
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x02FC.

rdParm
Specifies the logical brush.

CreateFontIndirect

```
struct {
    DWORD   rdSize;
    WORD    rdFunction;
    LOGFONT rdParm;
}
```

rdParm
Contains the following elements:

Element	Description
bmWidth	Bitmap width
bmHeight	Bitmap height
bmWidthBytes	Bytes per raster line
bmPlanes	Number of color planes
bmBitsPixel	Number of adjacent color bits that define a pixel
bmBits	Pointer to bit values
bits	Actual bits of pattern

CreatePatternBrush

3.0

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

The **CreatePatternBrush** record contains a device-independent bitmap suitable for playback on all devices.

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0142.

rdParm
Contains the following elements:

Element	Description
type	Bitmap type. This element may be either of these two values: BS_PATTERN—Brush is defined by a device-dependent bitmap through a call to the **CreatePatternBrush** function. BS_DIBPATTERN—Brush is defined by a device-independent bitmap through a call to the **CreateDIB-PatternBrush** function.

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x06FF.

rdParm
Specifies the region to be created.

DeleteObject

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm;
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x01F0.

rdParm
Specifies the index to the handle table for the object to be deleted.

Escape

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0626.

rdParm
Contains the following elements:

Element	Description
dxarray	Optional word array of intercharacter distances.

Polygon

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0324.

rdParm
Contains the following elements:

Element	Description
count	Number of points
list of points	List of individual points

PolyPolygon

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0538.

rdParm
Contains the following elements:

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x012C.

rdParm
Specifies the index to the handle table for the region being selected.

SelectObject

```
struct{
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm;
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x012D.

rdParm
Specifies the index to the handle table for the object being selected.

SelectPalette

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm;
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0234.

rdParm
Specifies the index to the handle table for the logical palette being selected.

SetPaletteEntries

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0037.

rdParm
Contains the following elements:

Element	Description
start	First entry to be set in the palette
numentries	Number of entries to be set in the palette
entries	**PALETTEENTRY** blocks (For a description of the **PALETTEENTRY** structure, see the *Microsoft Windows Programmer's Reference, Volume 3.*)

StretchBlt

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

The **StretchBlt** record contains a device-dependent bitmap that may not be suitable for playback on all devices.

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0B23.

rdParm
Contains the following elements:

Element	Description
raster op	Low-order word of the raster operation
raster op	High-order word of the raster operation
SYE	Source y-extent
SXE	Source x-extent
SY	Y-coordinate of the source origin
SX	X-coordinate of the source origin
DYE	Destination y-extent
DXE	Destination x-extent
DY	Y-coordinate of the destination origin
DX	X-coordinate of the destination origin
BitmapInfo	**BITMAPINFO** structure (For a description of the **BITMAPINFO** structure, see the *Microsoft Windows Programmer's Reference, Volume 3.*)
bits	Actual device-independent bitmap bits

StretchDIBits

```
struct {
    DWORD rdSize;
    WORD  rdFunction;
    WORD  rdParm[];
}
```

Members

rdSize
Specifies the record size, in words.

rdFunction
Specifies the GDI function number 0x0F43.

rdParm
Contains the following elements:

Element	Description
dwRop	Raster operation to be performed
Usag	Flag indicating whether the bitmap color table contains RGB values or indexes to the currently realized logical palette
srcYExt	Height of the source in the bitmap
srcXExt	Width of the source in the bitmap
srcY	Y-coordinate of the origin of the source in the bitmap

```
struct FONTINFO {
        WORD  dfVersion;
        DWORD dfSize;
        char  dfCopyright[60];
        WORD  dfType;
        WORD  dfPoints;
        WORD  dfVertRes;
        WORD  dfHorizRes;
        WORD  dfAscent;
        WORD  dfInternalLeading;
        WORD  dfExternalLeading;
        BYTE  dfItalic;
        BYTE  dfUnderline;
        BYTE  dfStrikeOut;
        WORD  dfWeight;
        BYTE  dfCharSet;
        WORD  dfPixWidth;
        WORD  dfPixHeight;
        BYTE  dfPitchAndFamily;
        WORD  dfAvgWidth;
        WORD  dfMaxWidth;
        BYTE  dfFirstChar;
        BYTE  dfLastChar;
        BYTE  dfDefaultChar;
        BYTE  dfBreakChar;
        WORD  dfWidthBytes;
        DWORD dfDevice;
        DWORD dfFace;
        DWORD dfBitsPointer;
        DWORD dfBitsOffset;
        BYTE  dfReserved;
        DWORD dfFlags;
        WORD  dfAspace;
        WORD  dfBspace;
        WORD  dfCspace;
        WORD  dfColorPointer;
        DWORD dfReserved1;
        WORD  dfCharTable[];
};
```

Following are the members of the **FONTINFO** structure:

dfVersion
Specifies the version (0x0200 or 0x0300) of the file.

dfSize
Specifies the total size of the file, in bytes.

dfCopyright
Specifies copyright information.

dfUnderline
Specifies whether the character-definition data represents an underlined font. If the flag is set, the low-order bit is 1. All other bits are zero.

dfStrikeOut
Specifies whether the character-definition data represents a strikeout font. If the flag is set, the low-order bit is 1. All other bits are zero.

dfWeight
Specifies the weight of the characters in the character-definition data, on a scale of 1 through 1000. A **dfWeight** value of 400 specifies a regular weight.

dfCharSet
Specifies the character set defined by this font.

dfPixWidth
Specifies the width of the grid on which a vector font was digitized. For raster fonts, if the **dfPixWidth** member is nonzero, it represents the width for all the characters in the bitmap. If the member is zero, the font has variable-width characters whose widths are specified in the array for the **dfCharTable** member.

dfPixHeight
Specifies the height of the character bitmap for raster fonts or the height of the grid on which a vector font was digitized.

dfPitchAndFamily
Specifies the pitch and font family. If the font is variable pitch, the low bit is set. The four high bits give the family name of the font. Font families describe the general look of a font. They identify fonts when the exact name is not available. The font families are described as follows:

Family	Description
FF_DONTCARE	Unknown.
FF_ROMAN	Proportionally spaced fonts with serifs.
FF_SWISS	Proportionally spaced fonts without serifs.
FF_MODERN	Fixed-pitch fonts.
FF_SCRIPT	Cursive or script fonts. (Both are designed to look similar to handwriting. Script fonts have joined letters; cursive fonts do not.)
FF_DECORATIVE	Novelty fonts.

dfAvgWidth
Specifies the width of characters in the font. For fixed-pitch fonts, this value is the same as the value for the **dfPixWidth** member. For variable-pitch fonts, it is the width of the character "X".

dfMaxWidth
Specifies the maximum pixel width of any character in the font. For fixed-pitch fonts, this value is the same as the value of the **dfPixWidth** member.

the strokes for each character of the font. The value of the **dfBitsOffset** member must be even.

dfReserved
Not used.

dfFlags
Specifies the bit flags that define the format of the glyph bitmap, as follows:

Pitch value	Address
DFF_FIXED	0x0001
DFF_PROPORTIONAL	0x0002
DFF_ABCFIXED	0x0004
DFF_ABCPROPORTIONAL	0x0008
DFF_1COLOR	0x0010
DFF_16COLOR	0x0020
DFF_256COLOR	0x0040
DFF_RGBCOLOR	0x0080

dfAspace
Specifies the global A space, if any. The value of the **dfAspace** member is the distance from the current position to the left edge of the bitmap.

dfBspace
Specifies the global B space, if any. The value of the **dfBspace** member is the width of the character.

dfCspace
Specifies the global C space, if any. The value of the **dfCspace** member is the distance from the right edge of the bitmap to the new current position. The increment of a character is the sum of the A, B, and C spaces. These spaces apply to all glyphs, including DFF_ABCFIXED.

dfColorPointer
Specifies the offset to the color table for color fonts, if any. The format of the bits is like a device-independent bitmap (DIB), but without the header. (That is, the characters are not split into disjoint bytes; instead, they are left intact.) If no color table is needed, this entry is NULL.

dfReserved1
Not used.

dfCharTable
Specifies an array of entries for raster, fixed-pitch vector, and proportionally spaced vector fonts, as follows:

This continues until the first "column" is completely defined. The subsequent byte contains the next 8 bits of the first scan line, padded with zeros on the right if necessary (and so on, down through the second "column"). If the glyph is quite narrow, each scan line is covered by one byte, with bits set to zero as necessary for padding. If the glyph is very wide, a third or even fourth set of bytes can be present.

Character bitmaps must be stored contiguously and arranged in ascending order. The bytes for a 12-pixel by 14-pixel "A" character, for example, are given in two sets, because the character is less than 17 pixels wide:

```
00 06 09 10 20 20 20 3F 20 20 20 00 00 00
00 00 00 80 40 40 40 C0 40 40 40 00 00 00
```

Note that in the second set of bytes, the second digit of the byte is always zero. The zeros correspond to the thirteenth through sixteenth pixels on the right side of the character, which are not used by this character bitmap.

4.3 Version-Specific Glyph Tables

The **dfCharTable** member for Windows 2.*x* has a **GlyphEntry** structure with the following format:

```
GlyphEntry      struc
geWidth                 dw      ?  ; width of char bitmap, pixels
geOffset                dw      ?  ; pointer to the bits
GlyphEntry      ends
```

The **dfCharTable** member in Windows 3.0 and later is dependent on the format of the glyph bitmap. The only formats supported are DFF_FIXED and DFF_PROPORTIONAL.

```
DFF_FIXED
DFF_PROPORTIONAL

GlyphEntry      struc
geWidth                 dw      ?  ; width of char bitmap, pixels
geOffset                dd      ?  ; pointer to the bits
GlyphEntry      ends

DFF_ABCFIXED
DFF_ABCPROPORTIONAL
```

```
struct tagGROUPHEADER {
    char  cIdentifier[4];
    WORD  wCheckSum;
    WORD  cbGroup;
    WORD  nCmdShow;
    RECT  rcNormal;
    POINT ptMin;
    WORD  pName;
    WORD  wLogPixelsX;
    WORD  wLogPixelsY;
    WORD  wBitsPerPixel;
    WORD  wPlanes;
    WORD  cItems;
    WORD  rgiItems[cItems];
};
```

Following are the members in the **GROUPHEADER** structure:

cIdentifier
Identifies an array of 4 characters. If the file is a valid group file, this array must contain the string "PMCC".

wCheckSum
Specifies the negative sum of all words in the file (including the value specified by the **wCheckSum** member).

cbGroup
Specifies the size of the group file, in bytes.

nCmdShow
Specifies whether Program Manager should display the group in minimized, normal, or maximized form. This member can be one of the following values:

Value	Flag
0x00	SW_HIDE
0x01	SW_SHOWNORMAL
0x02	SW_SHOWMINIMIZED
0x03	SW_SHOWMAXIMIZED
0x04	SW_SHOWNOACTIVATE
0x05	SW_SHOW
0x06	SW_MINIMIZE
0x07	SW_SHOWMINNOACTIVATE
0x08	SW_SHOWNA
0x09	SW_RESTORE

rcNormal
Specifies the coordinates of the group window (the window in which the group icons appear). It is a rectangular structure.

iIcon
Specifies the index value for an icon. This value indicates the position of the icon in an executable file.

cbResource
Specifies the count of bytes in the icon resource, which appears in the executable file for the application.

cbANDPlane
Specifies the count of bytes in the AND mask for the icon.

cbXORPlane
Specifies the count of bytes in the XOR mask for the icon.

pHeader
Specifies an offset from the beginning of the group file to the resource header for the icon.

pANDPlane
Specifies an offset from the beginning of the group file to the AND mask for the icon.

pXORPlane
Specifies an offset from the beginning of the group file to the XOR mask for the icon.

pName
Specifies an offset from the beginning of the group file to a string that specifies the item name.

pCommand
Specifies an offset from the beginning of the group file to a string that specifies the name of the executable file containing the application and the icon resource(s).

pIconPath
Specifies an offset from the beginning of the group file to a string that specifies the path where the executable file is located. This path can be used to extract icon data from an executable file.

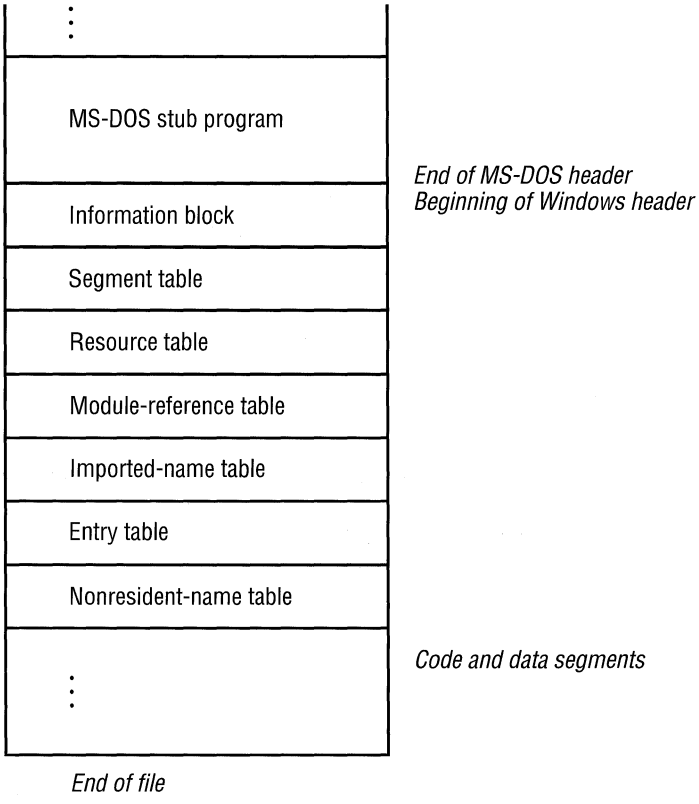
5.2.3 Tag Data

The tag data contains general information used to display the Program Item Properties dialog box. The **TAGDATA** structure has the following form:

```
struct tagTAGDATA{
    WORD wID;
    WORD wItem;
    WORD cb;
    BYTE rgb[1];
};
```

6.2 Windows Header

The Windows (new-style) executable-file header contains information that the loader requires for segmented executable files. This information includes the linker version number, data specified by the linker, data specified by the resource compiler, tables of segment data, tables of resource data, and so on. The following illustration shows the Windows executable-file header:



The following sections describe the entries in the Windows executable-file header.

Location	Description
	Bit **Meaning**
	15 If this bit is set, the executable file is a library module. If bit 15 is set, the CS:IP registers point to an initialization procedure called with the value in the AX register equal to the module handle. The initialization procedure must execute a far return to the caller. If the procedure is successful, the value in AX is nonzero. Otherwise, the value in AX is zero. The value in the DS register is set to the library's data segment if **SINGLEDATA** is set. Otherwise, DS is set to the data segment of the application that loads the library.
0Eh	Specifies the automatic data segment number. (0Eh is zero if the **SINGLEDATA** and **MULTIPLEDATA** bits are cleared.)
10h	Specifies the initial size, in bytes, of the local heap. This value is zero if there is no local allocation.
12h	Specifies the initial size, in bytes, of the stack. This value is zero if the SS register value does not equal the DS register value.
14h	Specifies the segment:offset value of CS:IP.
18h	Specifies the segment:offset value of SS:SP. The value specified in SS is an index to the module's segment table. The first entry in the segment table corresponds to segment number 1. If SS addresses the automatic data segment and SP is zero, SP is set to the address obtained by adding the size of the automatic data segment to the size of the stack.
1Ch	Specifies the number of entries in the segment table.
1Eh	Specifies the number of entries in the module-reference table.
20h	Specifies the number of bytes in the nonresident-name table.
22h	Specifies a relative offset from the beginning of the Windows header to the beginning of the segment table.
24h	Specifies a relative offset from the beginning of the Windows header to the beginning of the resource table.
26h	Specifies a relative offset from the beginning of the Windows header to the beginning of the resident-name table.
28h	Specifies a relative offset from the beginning of the Windows header to the beginning of the module-reference table.
2Ah	Specifies a relative offset from the beginning of the Windows header to the beginning of the imported-name table.

6.2.2 Segment Table

The segment table contains information that describes each segment in an executable file. This information includes the segment length, segment type, and segment-relocation data. The following list summarizes the values found in the segment table (the locations are relative to the beginning of each entry):

Location	Description
00h	Specifies the offset, in sectors, to the segment data (relative to the beginning of the file). A value of zero means no data exists.
02h	Specifies the length, in bytes, of the segment, in the file. A value of zero indicates that the segment length is 64K, unless the selector offset is also zero.
04h	Specifies flags that describe the contents of the executable file. This value can be one or more of the following:

Bit	Meaning
0	If this bit is set, the segment is a data segment. Otherwise, the segment is a code segment.
1	If this bit is set, the loader has allocated memory for the segment.
2	If this bit is set, the segment is loaded.
3	Reserved.
4	If this bit is set, the segment type is **MOVEABLE**. Otherwise, the segment type is **FIXED**.
5	If this bit is set, the segment type is **PURE** or **SHAREABLE**. Otherwise, the segment type is **IMPURE** or **NONSHAREABLE**.
6	If this bit is set, the segment type is **PRELOAD**. Otherwise, the segment type is **LOADONCALL**.
7	If this bit is set and the segment is a code segment, the segment type is **EXECUTEONLY**. If this bit is set and the segment is a data segment, the segment type is **READONLY**.
8	If this bit is set, the segment contains relocation data.
9	Reserved.
10	Reserved.
11	Reserved.
12	If this bit is set, the segment is discardable.
13	Reserved.

6.2.3.1 Type Information

The **TYPEINFO** structure has the following form:

```
typedef struct _TYPEINFO {
    WORD        rtTypeID;
    WORD        rtResourceCount;
    DWORD       rtReserved;
    NAMEINFO    rtNameInfo[];
} TYPEINFO;
```

Following are the members in the **TYPEINFO** structure:

rtTypeID
Specifies the type identifier of the resource. This integer value is either a resource-type value or an offset to a resource-type name. If the high bit in this member is set (0x8000), the value is one of the following resource-type values:

Value	Resource type
RT_ACCELERATOR	Accelerator table
RT_BITMAP	Bitmap
RT_CURSOR	Cursor
RT_DIALOG	Dialog box
RT_FONT	Font component
RT_FONTDIR	Font directory
RT_GROUP_CURSOR	Cursor directory
RT_GROUP_ICON	Icon directory
RT_ICON	Icon
RT_MENU	Menu
RT_RCDATA	Resource data
RT_STRING	String table

If the high bit of the value in this member is not set, the value represents an offset, in bytes relative to the beginning of the resource table, to a name in the **rscResourceNames** member.

rtResourceCount
Specifies the number of resources of this type in the executable file.

rtReserved
Reserved.

rtNameInfo
Specifies an array of **NAMEINFO** structures containing information about individual resources. The **rtResourceCount** member specifies the number of structures in the array.

6.2.4 Resident-Name Table

The resident-name table contains strings that identify exported functions in the executable file. As the name implies, these strings are resident in system memory and are never discarded. The resident-name strings are case-sensitive and are not null-terminated. The following list summarizes the values found in the resident-name table (the locations are relative to the beginning of each entry):

Location	Description
00h	Specifies the length of a string. If there are no more strings in the table, this value is zero.
01h – *xx*h	Specifies the resident-name text. This string is case-sensitive and is not null-terminated.
*xx*h + 01h	Specifies an ordinal number that identifies the string. This number is an index into the entry table.

The first string in the resident-name table is the module name.

6.2.5 Module-Reference Table

The module-reference table contains offsets for module names stored in the imported-name table. Each entry in this table is 2 bytes long.

6.2.6 Imported-Name Table

The imported-name table contains the names of modules that the executable file imports. Each entry contains two parts: a single byte that specifies the length of the string and the string itself. The strings in this table are not null-terminated.

6.2.7 Entry Table

The entry table contains bundles of entry points from the executable file (the linker generates each bundle). The numbering system for these ordinal values is 1-based—that is, the ordinal value corresponding to the first entry point is 1.

The linker generates the densest possible bundles under the restriction that it cannot reorder the entry points. This restriction is necessary because other executable files may refer to entry points within a given bundle by their ordinal values.

The entry-table data is organized by bundle, each of which begins with a 2-byte header. The first byte of the header specifies the number of entries in the bundle (a value of 00h designates the end of the table). The second byte specifies whether the corresponding segment is movable or fixed. If the value in this byte is 0FFh, the segment is movable. If the value in this byte is 0FEh, the entry does not refer

6.2.8 Nonresident-Name Table

The nonresident-name table contains strings that identify exported functions in the executable file. As the name implies, these strings are not always resident in system memory and are discardable. The nonresident-name strings are case-sensitive; they are not null-terminated. The following list summarizes the values found in the nonresident-name table (the specified locations are relative to the beginning of each entry):

Location	Description
00h	Specifies the length, in bytes, of a string. If this byte is 00h, there are no more strings in the table.
01h – *xx*h	Specifies the nonresident-name text. This string is case-sensitive and is not null-terminated.
xx + 01h	Specifies an ordinal number that is an index to the entry table.

The first name that appears in the nonresident-name table is the module description string (which was specified in the module-definition file).

6.3 Code Segments and Relocation Data

Code and data segments follow the Windows header. Some of the code segments may contain calls to functions in other segments and may, therefore, require relocation data to resolve those references. This relocation data is stored in a relocation table that appears immediately after the code or data in the segment. The first 2 bytes in this table specify the number of relocation items the table contains. A relocation item is a collection of bytes specifying the following information:

- Address type (segment only, offset only, segment and offset)
- Relocation type (internal reference, imported ordinal, imported name)
- Segment number or ordinal identifier (for internal references)
- Reference-table index or function ordinal number (for imported ordinals)
- Reference-table index or name-table offset (for imported names)

Each relocation item contains 8 bytes of data, the first byte of which specifies one of the following relocation-address types:

Value	Meaning
0	Low byte at the specified offset
2	16-bit selector
3	32-bit pointer
5	16-bit offset

Each icon-directory resource must have a corresponding entry in the resource table of the executable file. This means the resource table must contain a **TYPEINFO** structure in which the **rscTypeID** member is set to the RT_GROUP_ICON value.

7.3 Cursor Resource

A cursor resource is nearly identical in format to a cursor image in a cursor-resource file. The resource contains the cursor hot spot as well as the cursor-image header, color table, and XOR and AND masks. The x- and y-coordinates for the cursor hot spot (both 16-bit values) appear first in the resource, immediately followed by the cursor-image header. For more information about the cursor-image format, see Chapter 1, "Graphics File Formats."

Each cursor resource must have a corresponding entry in the resource table of the executable file. This means the resource table must contain a **TYPEINFO** structure in which the **rscTypeID** member is set to the RT_CURSOR value.

7.4 Cursor-Directory Resource

A cursor-directory resource is nearly identical in format to a cursor directory in a cursor-resource file. The resource specifies the number of cursor images associated with this resource, as well as the dimensions of the images, but it does not include the hot-spot data. Furthermore, the last member of the **ICONDIRENTRY** structure (**dwImageOffset**) is replaced with a 16-bit value that specifies the resource-table index of the corresponding cursor-image resource.

In an executable file, the **CURSORDIRENTRY** structure has the following form:

```
typedef struct _CURSORDIRENTRY {
    WORD  wWidth;
    WORD  wHeight;
    WORD  wPlanes;
    WORD  wBitCount;
    DWORD lBytesInRes;
    WORD  wImageIndex;
} CURSORDIRENTRY;
```

Following are the members in the **CURSORDIRENTRY** structure:

wWidth
Specifies the width of the cursor, in pixels.

wHeight
Specifies the height of the cursor, in pixels.

7.5.2 Pop-up Menu Item

A menu resource contains data for each pop-up item in a menu. The first 16 bits indicate whether the item is grayed, inactive, checked, and so on. This data also includes a string that appears in the rectangle corresponding to that item. A **PopupMenuItem** structure has the following form:

```
struct PopupMenuItem {
    WORD fItemFlags;
    char szItemText[];
};
```

Following are the members in the **PopupMenuItem** structure:

fItemFlags
Specifies menu-item information. This member can have one or more of the following values:

Value	Meaning
MF_GRAYED	Item is grayed.
MF_DISABLED	Item is inactive.
MF_CHECKED	Item can be checked.
MF_POPUP	Item is a popup (must be specified for pop-up items).
MF_MENUBARBREAK	Item is a menu-bar break.
MF_MENUBREAK	Item is a menu break.
MF_END	Item ends the menu.

szItemText
Specifies a null-terminated string that appears in the menu and identifies the menu item. There is no fixed limit on the size of this string.

7.5.3 Normal Menu Item

A normal menu item is very similar to a pop-up menu item, except that it has an additional menu identifier. A **NormalMenuItem** structure has the following form:

```
struct NormalMenuItem {
    WORD fItemFlags;
    WORD wMenuID;
    char szItemText[];
};
```

```
POPUP ITEM
    NORMAL ITEM
    NORMAL ITEM
    .
    .
    .
    NORMAL ITEM
    POPUP ITEM
        NORMAL ITEM
        NORMAL ITEM
        NORMAL ITEM
        POPUP ITEM (fItemFlags contains the MF_END constant)
            NORMAL ITEM
            NORMAL ITEM (fItemFlags contains the MF_END constant)
    NORMAL ITEM (fItemFlags contains the MF_END constant)
```

Note that, although the pop-up menu item has its own terminating item, the terminating item for the entire menu is again a normal menu item.

7.6 Dialog Box Resource

A dialog box resource contains a dialog box header and data for each control within the dialog box.

Each entry in the executable file's resource table contains a member that identifies the resource type. The RT_DIALOG constant identifies a dialog box resource.

7.6.1 Dialog Box Header

The dialog box header contains general dialog box data, such as the dialog box window style, the number of controls in the dialog box, the coordinates of the upper-left corner of the box, the width and height of the box, the name of the menu to be displayed, and so on. The **DialogBoxHeader** structure has the following form:

```
struct DialogBoxHeader {
    DWORD lStyle;
    BYTE  bNumberOfItems;
    WORD  x;
    WORD  y;
    WORD  cx;
    WORD  cy;
    char  szMenuName[];
    char  szClassName[];
    char  szCaption[];
    WORD  wPointSize;    /* only if DS_SETFONT */
    char  szFaceName[];  /* only if DS_SETFONT */
};
```

wPointSize
Specifies the point size of a font that is unique to the dialog box. (This member is present only if the DS_SETFONT flag is set by the **lStyle** member.)

szFaceName
Specifies the typeface name of a dialog box font. This array must contain a null-terminated string. (This member is present only if the DS_SETFONT flag is set by the **lStyle** member.)

7.6.2 Control Data

A dialog box resource contains data for each control in a given dialog box. This data contains the coordinates of the upper-left corner of the control, the dimensions of the control, a control identifier, and so on. A **ControlData** structure has the following form:

```
struct ControlData {
    WORD  x;
    WORD  y;
    WORD  cx;
    WORD  cy;
    WORD  wID;
    DWORD lStyle;
    union
    {
        BYTE class;     /* if (class & 0x80) */
        char szClass[]; /* otherwise         */
    } ClassID;
    szText;
};
```

Following are the members in the **ControlData** structure:

x
Specifies the x-coordinate of the upper-left corner of the control.

y
Specifies the y-coordinate of the upper-left corner of the control.

cx
Specifies the width of the control, in horizontal dialog box units. For a definition of these units, see the **DialogBoxHeader** structure in the preceding section.

cy
Specifies the height of the control, in vertical dialog box units. For a definition of these units, see the **DialogBoxHeader** structure in the preceding section.

wID
Identifies the control.

7.8 Font Resource

A font resource consists of two parts: a directory and its components. The font-directory data describes all the fonts in a resource. This data includes a value specifying the number of fonts in the resource and a table of metrics for each of these fonts. The font-component data describes a single font in the resource. There is one component for each of the fonts in the resource. The component data is identical to the data found in a Windows font file (.FNT).

Each entry in the executable file's resource table contains a member that identifies the resource type. The RT_FONTDIR and RT_FONT constants identify a font directory and a font component, respectively.

7.8.1 Font-Directory Data

Font-directory data consists of a font count and one or more font directory entries.

7.8.1.1 Font Count

The font count is an integer that specifies the number of fonts in the resource. This value also corresponds to the number of font directories and font components.

7.8.1.2 Font Directory

The font directory is a collection of font metrics for a particular font. These metrics specify the point size for the font, aspect ratio, stroke width, and so on. The **FontDirEntry** structure has the following form:

```
struct FontDirEntry {
        WORD  fontOrdinal;
        WORD  dfVersion;
        DWORD dfSize;
        char  dfCopyright[60];
        WORD  dfType;
        WORD  dfPoints;
        WORD  dfVertRes;
        WORD  dfHorizRes;
        WORD  dfAscent;
        WORD  dfInternalLeading;
        WORD  dfExternalLeading;
        BYTE  dfItalic;
        BYTE  dfUnderline;
        BYTE  dfStrikeOut;
        WORD  dfWeight;
        BYTE  dfCharSet;
        WORD  dfPixWidth;
        WORD  dfPixHeight;
```

Data structure	Contents
Kerning-pair data	An identifier for each character in the pair of kerned characters, and a kerning value
Track-kerning data	Additional kerning data

For a complete description of Windows font files, see the Microsoft Windows Device Development Kit documentation.

7.9 String-Table Resources

A string table consists of one or more separate resources, each containing exactly 16 strings. The maximum length of each string is 255 bytes. One or more strings in a block can be null or empty. The first byte in the string specifies the number of characters in the string. (For null or empty strings, the first byte contains the value zero.)

Windows uses a 16-bit identifier to locate a string in a string-table resource. Bits 4 through 15 specify the block in which the string appears; bits 0 through 3 specify the location of that string relative to the beginning of the block.

Each entry in an executable file's resource table contains a member that identifies the resource type. The RT_STRING constant identifies a string table.

7.10 Accelerator Resource

An accelerator resource contains one or more accelerator entries.

Each entry in an executable file's resource table contains a member that identifies the resource type. The RT_ACCELERATOR constant identifies an accelerator resource.

The accelerator entry is a 5-byte entry with the following form:

```
struct AccelTableEntry {
        BYTE fFlags;
        WORD wEvent;
        WORD wId;
};
```

Following are the members in the **AccelTableEntry** structure:

fFlags
Specifies accelerator characteristics. It can be one or more of the following values:

Microsoft® Windows™

Version 3.1

Multimedia Programmer's Reference

For the Microsoft Windows Operating System

Microsoft Corporation

Contents

Chapter 1 Introduction

Chapter 2 Function Overview

Chapter 4 Message Overview

Chapter 5 Message Directory

Chapter 6 Data Types and Structures

Chapter 8 Multimedia File Formats

Chapter 1
Introduction

This manual provides reference information for the multimedia portions of the application programming interface (API) of Microsoft® Windows™ Operating System 3.1. The multimedia APIs provide support for audio, media control, multimedia file I/O, enhanced timer services, and joystick input.

The multimedia APIs include functions, messages, data structures, data types, and file formats to add multimedia support to your Windows applications. For information on other Windows APIs, see the *Microsoft Windows Programmer's Reference, Volume 2: API Reference.*

Windows Multimedia Features

Windows offers services you can use to add features like sound recording and playback, MIDI music, and external device control to your applications. Windows provides the following multimedia services:

- Audio—The audio services provide a device-independent interface to computer-audio hardware, providing sound for multimedia applications.
- The Media Control Interface (MCI)—MCI provides a high-level generalized interface to control media devices such as audio hardware, movie players, and videodisc and videotape players.
- Multimedia File I/O—The multimedia file I/O services provide buffered and unbuffered file I/O, as well as support for standard Resource Interchange File Format (RIFF) files. These services are extensible with custom I/O procedures that can be shared among applications.
- Joystick and timer—These services provide support for joysticks and high-resolution event timing.

Message Names

Message names, like function names, begin with a prefix. Related messages are grouped together with a common prefix. An underscore character (_) follows the prefix in each message name. One or more words describing the purpose of the message appear after the underscore. Message names use only uppercase letters.

The following is an example of a message name:

WOM_CLOSE

The prefix (*WOM*) indicates that the message is a waveform output message. The descriptive portion of the message (*CLOSE*) indicates the purpose of the message. This message is sent whenever a waveform output device is closed.

Parameter Names

Most parameter and local-variable names consist of a lowercase prefix followed by one or more capitalized words. The prefix indicates the general type of the parameter, while the words that follow describe the contents of the parameter. The standard prefixes used in parameter and variable names are defined as follows:

Prefix	Description
b	Boolean (a non-zero value specifies TRUE, zero specifies FALSE)
ch	Character (a one-byte value)
dw	Long (32-bit) unsigned integer
h	Handle
l	Long (32-bit) integer
lp	Far pointer
np	Near pointer
pt	*x* and *y* coordinates packed into an unsigned 32-bit integer
rgb	An RGB color value packed into a 32-bit integer
w	Short (16-bit) unsigned integer

Note If no lowercase prefix is given, the parameter is a short integer with a descriptive name.

- Chapter 7, "MCI Command Strings," describes command strings for the Media Control Interface (MCI). It describes how to use MCI command strings and describes each command string recognized by MCI. Commands are grouped by device type.
- Chapter 8, "Multimedia File Formats," describes the multimedia file formats.
- Appendix A, "MCI Command String Syntax Summary," presents a summary of the syntax of the MCI command strings.
- Appendix B, "Manufacturer ID and Product ID Lists," lists the constants that identify multimedia product manufacturers and products used with Windows.

Conventions

The following section explains the document conventions used throughout this manual:

Type Style	Used For
bold	Bold letters indicate a specific term intended to be used literally: functions (such as **waveOutGetNumDevs**), messages (such as **WIM_OPEN**), and structure fields (such as **dwReturn**). You must enter these terms exactly as shown.
italic	Words in italics indicate a placeholder; you are expected to provide the actual value. For example, the following syntax for the **timeGetSystemTime** function indicates that you must substitute values for the *lpTime* and *wSize* parameters: **timeGetSystemTime**(*lpTime,wSize*)
`monospace`	Code examples are displayed in a monospaced typeface.
brackets [ ]	Optional items.
Horizontal ellipsis ...	An ellipsis shows that one or more copies of the preceding item may occur. Brackets followed by an ellipsis means that the item enclosed within the brackets may occur zero or more times.
Angle brackets < >	Indicates the name and position of a field within a file format definition.
Arrow →	In a file format definition, the item to the left of the arrow is equivalent to the item to the right.

Chapter 2
Function Overview

This chapter provides a topical overview of the multimedia functions in Windows. The functions are organized into the following categories, some of which contain smaller groups of related functions:

- High-level audio services
- Low-level waveform audio services
- Low-level MIDI audio services
- Auxiliary audio device services
- File I/O services
- Media Control Interface (MCI) services
- Joystick services
- Timer services
- Debugging services

For full descriptions of the multimedia functions, see the alphabetical listing in Chapter 3, "Function Directory."

Querying Waveform Devices

Before playing or recording a waveform, you must determine the capabilities of the waveform hardware present in the system. Use the following functions to retrieve the number of waveform devices and the capabilities of each device:

waveInGetNumDevs
Retrieves the number of waveform input devices present in the system.

waveInGetDevCaps
Retrieves the capabilities of a given waveform input device.

waveOutGetNumDevs
Retrieves the number of waveform output devices present in the system.

waveOutGetDevCaps
Retrieves the capabilities of a given waveform output device.

Opening and Closing Waveform Devices

You must open a device before you can begin waveform playback or recording. Once you finish using a device, you must close it so that it will be available to other applications. Use the following functions to open and close waveform devices:

waveInOpen
Opens a waveform input device for recording.

waveInClose
Closes a specified waveform input device.

waveOutOpen
Opens a waveform output device for playback.

waveOutClose
Closes a specified waveform output device.

waveInPrepareHeader
Informs the waveform input device driver that the given data buffer should be prepared for recording.

waveInUnprepareHeader
Informs the waveform input device driver that the preparation performed on the given data buffer can be cleaned up.

Getting the Current Position of Waveform Devices

While playing or recording waveform audio, you can query the device for the current playback or recording position. Use the following functions to determine the current position of a waveform device:

waveInGetPosition
Retrieves the current recording position of a waveform input device.

waveOutGetPosition
Retrieves the current playback position of a waveform output device.

Controlling Waveform Playback

Waveform playback begins as soon as you begin sending data to the waveform output device. Use the following functions to pause, restart, or stop playback and to break loops on a waveform device:

waveOutBreakLoop
Breaks a loop on a waveform output device.

waveOutPause
Pauses playback on a waveform output device.

waveOutRestart
Resumes playback on a paused waveform output device.

waveOutReset
Stops playback on a waveform output device. Marks all pending data blocks as done.

Changing Playback Volume

Some waveform output devices support changes to the playback volume level. Use these functions to query and set the volume level of waveform output devices:

waveOutGetVolume
Queries the current volume level of a waveform output device.

waveOutSetVolume
Sets the volume level of a waveform output device.

Sending Custom Messages to Waveform Drivers

The following functions let you send messages directly to waveform drivers:

waveInMessage
Sends a message directly to a waveform input device driver.

waveOutMessage
Sends a message directly to a waveform input device driver.

Handling Waveform Errors

Most of the low-level waveform audio functions return error codes. Use these functions to convert the error codes returned from waveform functions into a textual description of the error:

waveInGetErrorText
Retrieves a textual description of a specified waveform input error.

waveOutGetErrorText
Retrieves a textual description of a specified waveform output error.

Opening and Closing MIDI Devices

After getting the MIDI capabilities, you must open a MIDI device to play or record MIDI messages. After using the device, you should close it to make it available to other applications. Use the following functions to open and close MIDI devices:

midiInOpen
Opens a MIDI input device for recording.

midiInClose
Closes a specified MIDI input device.

midiOutOpen
Opens a MIDI output device for playback.

midiOutClose
Closes a specified MIDI output device.

Getting the Device ID of MIDI Devices

Using a MIDI device handle, you can retrieve the device ID for an open MIDI device. Use the following functions to get the device ID:

midiInGetID
Gets the device ID for a MIDI input device.

midiOutGetID
Gets the device ID for a MIDI output device.

Receiving MIDI Messages

Once you open a MIDI input device, you can begin receiving MIDI input. MIDI messages other than system exclusive messages are sent directly to a callback. To receive system exclusive messages, you must pass data buffers to the input device. These data buffers must be prepared before being sent to the device. Use the following messages to prepare system exclusive data buffers and pass these buffers to a MIDI input device:

midiInAddBuffer

Sends an input buffer for system exclusive messages to a specified MIDI input device. The buffer is sent back to the application when it is filled with system exclusive data.

midiInPrepareHeader

Informs a MIDI input device that the given data buffer should be prepared for recording.

midiInUnprepareHeader

Informs a MIDI input device that the preparation performed on the given data buffer can be cleaned up.

Controlling MIDI Input

When receiving MIDI input, you can control when the input starts and stops. Use the following functions to start and stop input on a MIDI input device:

midiInStart

Starts input on a MIDI input device.

midiInStop

Stops input on a MIDI input device.

midiInReset

Stops input on a MIDI input device. Marks all pending data buffers as being done.

Sending Custom Messages to MIDI Drivers

The following functions let you send messages directly to MIDI device drivers:

midiInMessage
Sends a message directly to a MIDI input device driver.

midiOutMessage
Sends a message directly to a MIDI output device driver.

Auxiliary Audio Services

Auxiliary audio devices are audio devices whose output is mixed with the output of waveform and MIDI synthesizer devices. Use the following functions to query the capabilities of auxiliary audio devices and to query and set their volume level:

auxGetDevCaps
Retrieves the capabilities of a given auxiliary audio device.

auxGetNumDevs
Retrieves the number of auxiliary audio devices present in a system.

auxGetVolume
Queries the volume level of an auxiliary audio device.

auxOutMessage
Sends a message to an auxiliary output device.

auxSetVolume
Sets the volume level of an auxiliary audio device.

Performing Buffered File I/O

Using the basic buffered file I/O services is very similar to using the unbuffered services. Specify the MMIO_ALLOCBUF option with the **mmioOpen** function to open a file for buffered I/O. The file I/O manager will maintain an internal buffer which is transparent to the application.

You can also change the size of the internal buffer, allocate your own buffer, and directly access a buffer for optimal I/O performance. Use the following functions for I/O buffer control and direct I/O buffer access:

mmioAdvance
Fills and/or flushes the I/O buffer of a file set up for direct I/O buffer access.

mmioFlush
Writes the contents of the I/O buffer to disk.

mmioGetInfo
Gets information about the file I/O buffer of a file opened for buffered I/O.

mmioSetBuffer
Changes the size of the I/O buffer, and allows applications to supply their own buffer.

mmioSetInfo
Changes information about the file I/O buffer of a file opened for buffered I/O.

Media Control Interface Services

The Media Control Interface (MCI) provides a high-level generalized interface for controlling both internal and external media devices. MCI uses device handlers to interpret and execute high-level MCI commands. Applications can communicate with MCI device handlers by sending messages or command strings.

MCI also provides macros for working with the time and position information encoded in a packed DWORD.

Communicating with MCI Devices

You can communicate with MCI devices using messages or command strings. Messages are used directly by MCI; MCI converts command strings into messages that it then sends to the device handler. Use these functions to send messages or command strings to MCI, to get the ID assigned to a device, and to get a textual description of an MCI error:

mciSendCommand
Sends a command message to MCI.

mciSendString
Sends a command string to MCI.

mciGetDeviceID
Returns the device ID assigned when the device was opened.

mciGetErrorString
Returns the error string corresponding to an MCI error return value.

mciSetYieldProc
Specifies a callback procedure to be called while an MCI device is completing a command specified with the wait flag.

mciGetYieldProc
Returns the current yield procedure for an MCI device.

Most of the MCI functionality is expressed in its command set. See Chapter 4, "Message Overview," and Chapter 5, "Message Directory," for an overview and reference to all MCI command messages. MCI command messages are prefixed with **MCI**.

In addition to its message-based interface, MCI has a string-based interface. Chapter 7, "MCI Command Strings," describes the MCI command strings.

MMSYSTEM.H also defines the following macros that combine separate time and position values into the packed DWORD format:

MCI_MAKE_HMS
Creates a DWORD time value in hours/minutes/seconds format from the given hours, minutes, and seconds values.

MCI_MAKE_MSF
Creates a DWORD time value in minutes/seconds/frames format from the given minutes, seconds, and frames values.

MCI_MAKE_TMSF
Creates a DWORD time value in tracks/minutes/seconds/frames format from the given tracks, minutes, seconds, and frames values.

Joystick Services

The joystick services provide support for up to two joystick devices. Use the following functions to get information about joystick devices, to control joystick sensitivity, and to receive messages related to joystick movement and button activity:

joyGetDevCaps
Returns the capabilities of a joystick device.

joyGetNumDevs
Returns the number of devices supported by the joystick driver.

joyGetPos
Returns the position and button state of a joystick.

joyGetThreshold
Returns the movement threshold of a joystick.

joyReleaseCapture
Releases the joystick captured with **joySetCapture**.

joySetCapture
Causes periodic joystick messages to be sent to a window.

joySetThreshold
Sets the movement threshold of a joystick.

Chapter 3
Function Directory

This chapter contains an alphabetical list of the Windows multimedia functions. For information about standard Windows functions, see the *Microsoft Windows Programmer's Reference, Volume 2: API Reference.*

For each function, this chapter lists the following items:

- The syntax for the function
- The purpose of the function
- A description of input parameters
- A description of return values
- Optional comments on using the function
- Optional cross references to other functions, messages, and data structures

This chapter also lists the multimedia macros for Windows. Macros are documented similarly to functions. Each description begins by identifying the routine as a function or a macro (for example, "This *function*..." or "This *macro*...").

Function Descriptions

This section lists the multimedia functions and macros. The functions and macros are presented in alphabetical order.

auxGetDevCaps

Syntax

UINT **auxGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified auxiliary output device to determine its capabilities.

Parameters

UINT *wDeviceID*

Identifies the auxiliary output device to be queried. Specify a valid device ID (see the following "Comments" section), or use the following constant:

AUX_MAPPER

Auxiliary audio mapper. The function will return an error if no auxiliary audio mapper is installed.

LPAUXCAPS *lpCaps*

Specifies a far pointer to an AUXCAPS structure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the AUXCAPS structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver failed to install.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use **auxGetNumDevs** to determine the number of auxiliary output devices present in the system.

See Also

auxGetNumDevs

Comments Not all devices support volume control. To determine whether the device supports volume control, use the AUXCAPS_VOLUME flag to test the **dwSupport** field of the **AUXCAPS** structure (filled by **auxGetDevCaps**).

To determine whether the device supports volume control on both the left and right channels, use the AUXCAPS_LRVOLUME flag to test the **dwSupport** field of the **AUXCAPS** structure (filled by **auxGetDevCaps**).

See Also **auxSetVolume**

auxOutMessage

Syntax DWORD **auxOutMessage**(*wDeviceID*, *msg*, *dw1*, *dw2*)

This function sends a message to an auxiliary output device. It also performs error checking on the device ID passed.

Parameters

UINT *wDeviceID*
Specifies the auxiliary output device to receive the message.

UINT *msg*
Specifies the message to send.

DWORD *dw1*
Specifies the first message parameter.

DWORD *dw2*
Specifies the second message parameter.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID
The device ID specifies an invalid or nonexistent device.

MMSYSERR_NODRIVER
The driver for the auxiliary device failed to install.

joyGetDevCaps

Syntax UINT **joyGetDevCaps**(*wJoyID*, *lpCaps*, *wSize*)

This function queries a joystick device to determine its capabilities.

Parameters

UINT *wJoyID*
Identifies the device to be queried. This value is either JOYSTICKID1 or JOYSTICKID2.

LPJOYCAPS *lpCaps*
Specifies a far pointer to a **JOYCAPS** data structure. This structure is filled with information about the capabilities of the joystick device.

UINT *wSize*
Specifies the size of the **JOYCAPS** structure.

Return Value Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER
The joystick driver is not present.

JOYERR_PARMS
The specified joystick device ID *wJoyID* is invalid.

Comments Use **joyGetNumDevs** to determine the number of joystick devices supported by the driver.

See Also **joyGetNumDevs**

joyGetNumDevs

Syntax UINT **joyGetNumDevs**()

This function returns the number of joystick devices supported by the system.

Parameters None.

Return Value Returns the number of joystick devices supported by the joystick driver. If no driver is present, the function returns zero.

Comments Use **joyGetPos** to determine whether a given joystick is actually attached to the system. The **joyGetPos** function returns a JOYERR_UNPLUGGED error code if the specified joystick is not connected.

See Also **joyGetDevCaps**, **joyGetPos**

Return Value Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER
The joystick driver is not present.

JOYERR_PARMS
The specified joystick device ID *wJoyID* is invalid.

Comments The movement threshold is the distance the joystick must be moved before a WM_JOYMOVE message is sent to a window that has captured the device. The threshold is initially zero.

See Also **joySetThreshold**

joyReleaseCapture

Syntax UINT **joyReleaseCapture**(*wJoyID*)

This function releases the capture set by **joySetCapture** on the specified joystick device.

Parameters UINT *wJoyID*
Identifies the joystick device to be released. This value is either JOYSTICKID1 or JOYSTICKID2.

Return Value Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER
The joystick driver is not present.

JOYERR_PARMS
The specified joystick device ID *wJoyID* is invalid.

See Also **joySetCapture**

joySetThreshold

Syntax UINT **joySetThreshold**(*wJoyID*, *wThreshold*)

This function sets the movement threshold of a joystick device.

Parameters UINT *wJoyID*
Identifies the joystick device. This value is either JOYSTICKID1 or JOYSTICKID2.

UINT *wThreshold*
Specifies the new movement threshold.

Return Value Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER
The joystick driver is not present.

JOYERR_PARMS
The specified joystick device ID *wJoyID* is invalid.

Comments The movement threshold is the distance the joystick must be moved before an MM_JOYMOVE message is sent to a window that has captured the device.

See Also **joyGetThreshold, joySetCapture**

MCI_HMS_HOUR

Syntax BYTE **MCI_HMS_HOUR**(*dwHMS*)

This macro returns the hours field from a DWORD argument containing packed HMS (hours, minutes, seconds) information.

Parameters DWORD *dwHMS*
Specifies the time in HMS format.

Return Value The return value is the hours field of the given argument.

Comments Time in HMS format is expressed as a DWORD with the least significant byte containing hours, the next least significant byte containing minutes, and the next least significant byte containing seconds. The most significant byte is unused.

See Also **MCI_HMS_MINUTE, MCI_HMS_SECOND, MCI_MAKE_HMS**

MCI_MAKE_HMS

Syntax DWORD **MCI_MAKE_HMS**(*hours*, *minutes*, *seconds*)

This macro returns a time value in HMS (hours, minutes, seconds) format from the given hours, minutes, and seconds values.

Parameters BYTE *hours*
Specifies the number of hours.

BYTE *minutes*
Specifies the number of minutes.

BYTE *seconds*
Specifies the number of seconds.

Return Value The return value is a DWORD value containing the time in packed HMS format.

Comments Time in HMS format is expressed as a DWORD with the least significant byte containing hours, the next least significant byte containing minutes, and the next least significant byte containing seconds. The most significant byte is unused.

See Also **MCI_HMS_HOUR**, **MCI_HMS_MINUTE**, **MCI_HMS_SECOND**

MCI_MAKE_MSF

Syntax DWORD **MCI_MAKE_MSF**(*minutes*, *seconds*, *frames*)

This macro returns a time value in MSF (minutes, seconds, frames) format from the given minutes, seconds, and frames values.

Parameters BYTE *minutes*
Specifies the number of minutes.

BYTE *seconds*
Specifies the number of seconds.

BYTE *frames*
Specifies the number of frames.

Return Value The return value is a DWORD value containing the time in packed MSF format.

Comments Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also **MCI_MSF_MINUTE**, **MCI_MSF_SECOND**, **MCI_MSF_FRAME**

MCI_MSF_MINUTE

Syntax BYTE **MCI_MSF_MINUTE**(*dwMSF*)

This macro returns the minutes field from a DWORD argument containing packed MSF (minutes, seconds, frames) information.

Parameters DWORD *dwMSF*

Specifies the time in MSF format.

Return Value The return value is the minutes field of the given argument.

Comments Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also **MCI_MSF_SECOND, MCI_MSF_FRAME, MCI_MAKE_MSF**

MCI_MSF_SECOND

Syntax BYTE **MCI_MSF_SECOND**(*dwMSF*)

This macro returns the seconds field from a DWORD argument containing packed MSF (minutes, seconds, frames) information.

Parameters DWORD *dwMSF*

Specifies the time in MSF format.

Return Value The return value is the seconds field of the given argument.

Comments Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also **MCI_MSF_MINUTE, MCI_MSF_FRAME, MCI_MAKE_MSF**

MCI_TMSF_SECOND

Syntax BYTE **MCI_TMSF_SECOND**(*dwTMSF*)

This macro returns the seconds field from a DWORD argument containing packed TMSF (tracks, minutes, seconds, frames) information.

Parameters DWORD *dwTMSF*

Specifies the time in TMSF format.

Return Value The return value is the seconds field of the given argument.

Comments Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also **MCI_TMSF_TRACK, MCI_TMSF_MINUTE, MCI_TMSF_FRAME, MCI_MAKE_TMSF**

MCI_TMSF_TRACK

Syntax BYTE **MCI_TMSF_TRACK**(*dwTMSF*)

This macro returns the tracks field from a DWORD argument containing packed TMSF (tracks, minutes, seconds, frames) information.

Parameters DWORD *dwTMSF*

Specifies the time in TMSF format.

Return Value The return value is the tracks field of the given argument.

Comments Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also **MCI_TMSF_MINUTE, MCI_TMSF_SECOND, MCI_TMSF_FRAME, MCI_MAKE_TMSF**

mciGetYieldProc

Syntax

YIELDPROC WINAPI,_**mciGetYieldProc**(*wDeviceID*, *lpdwYieldData*)

This function returns the address of the callback procedure associated with the mci WAIT flag; the callback procedure is called periodically while an MCI device waits for a command specified with the WAIT flag to complete.

Parameters

UINT *wDeviceID*

Specifies the ID of the MCI device being monitored while it performs an MCI command.

LPDWORD *lpdwYieldData*

Optionally specifies a buffer to hold the yield data passed to the function. If the parameter is NULL, it is ignored.

Return Value

Returns the current yield proc if it exists. Otherwise, returns NULL for an invalid device ID.

mciSendCommand

Syntax

DWORD **mciSendCommand**(*wDeviceID*, *wMessage*, *dwParam1*, *dwParam2*)

This function sends a command message to the specified MCI device.

Parameters

UINT *wDeviceID*

Specifies the device ID of the MCI device to receive the command. This parameter is not used with the **MCI_OPEN** command.

UINT *wMessage*

Specifies the command message.

DWORD *dwParam1*

Specifies flags for the command.

DWORD *dwParam2*

Specifies a pointer to a parameter block for the command.

Return Value

Returns zero if the function was successful. Otherwise, it returns error information. The low-order word of the returned DWORD is the error return value. If the error is device-specific, the high-order word contains the driver ID; otherwise the high-order word is zero.

To get a text description of **mciSendCommand** return values, pass the return value to **mciGetErrorString**.

MCIERR_DEVICE_TYPE_REQUIRED

The specified device cannot be found on the system. Check that the device is installed and the device name is spelled correctly.

MCIERR_DRIVER

The device driver exhibits a problem. Check with the device manufacturer about obtaining a new driver.

MCIERR_DRIVER_INTERNAL

The device driver exhibits a problem. Check with the device manufacturer about obtaining a new driver.

MCIERR_DUPLICATE_ALIAS

The specified alias is already used in this application. Use a unique alias.

MCIERR_EXTENSION_NOT_FOUND

The specified extension has no device type associated with it. Specify a device type.

MCIERR_EXTRA_CHARACTERS

You must enclose a string with quotation marks; characters following the closing quotation mark are not valid.

MCIERR_FILE_NOT_FOUND

The requested file was not found. Check that the path and filename are correct.

MCIERR_FILE_NOT_SAVED

The file was not saved. Make sure your system has sufficient disk space or has an intact network connection.

MCIERR_FILE_READ

A read from the file failed. Make sure the file is present on your system or that your system has an intact network connection.

MCIERR_FILE_WRITE

A write to the file failed. Make sure your system has sufficient disk space or has an intact network connection.

MCIERR_FLAGS_NOT_COMPATIBLE

The specified parameters cannot be used together.

MCIERR_MUST_USE_SHAREABLE
The device driver is already in use. You must specify the "shareable" parameter with each open command to share the device.

MCIERR_NO_ELEMENT_ALLOWED
The specified device does not use a filename.

MCIERR_NO_INTEGER
The parameter for this MCI command must be an integer value.

MCIERR_NO_WINDOW
There is no display window.

MCIERR_NONAPPLICABLE_FUNCTION
The specified MCI command sequence cannot be performed in the given order. Correct the command sequence; then, try again.

MCIERR_NULL_PARAMETER_BLOCK
A null parameter block was passed to MCI.

MCIERR_OUT_OF_MEMORY
Your system does not have enough memory for this task. Quit one or more applications to increase the available memory, then, try to perform the task again.

MCIERR_OUTOFRANGE
The specified parameter value is out of range for the specified MCI command.

MCIERR_SET_CD
The specified file or MCI device is inaccessible because the application cannot change directories.

MCIERR_SET_DRIVE
The specified file or MCI device is inaccessible because the application cannot change drives.

MCIERR_UNNAMED_RESOURCE
You cannot store an unnamed file. Specify a filename.

MCIERR_UNRECOGNIZED_COMMAND
The driver cannot recognize the specified command.

MCIERR_UNSUPPORTED_FUNCTION
The MCI device driver the system is using does not support the specified command.

Waveform Audio Errors

The following additional return values are defined for MCI waveform audio devices:

MCIERR_WAVE_INPUTSINUSE
: All waveform devices that can record files in the current format are in use. Wait until one of these devices is free; then, try again.

MCIERR_WAVE_INPUTSUNSUITABLE
: No installed waveform device can record files in the current format. Use the Drivers option from the Control Panel to install a suitable waveform recording device.

MCIERR_WAVE_INPUTUNSPECIFIED
: You can specify any compatible waveform recording device.

MCIERR_WAVE_OUTPUTSINUSE
: All waveform devices that can play files in the current format are in use. Wait until one of these devices is free; then, try again.

MCIERR_WAVE_OUTPUTSUNSUITABLE
: No installed waveform device can play files in the current format. Use the Drivers option from the Control Panel to install a suitable waveform device.

MCIERR_WAVE_OUTPUTUNSPECIFIED
: You can specify any compatible waveform playback device.

MCIERR_WAVE_SETINPUTINUSE
: The current waveform device is in use. Wait until the device is free; then, try again to set the device for recording.

MCIERR_WAVE_SETINPUTUNSUITABLE
: The device you are using to record a waveform cannot recognize the data format.

MCIERR_WAVE_SETOUTPUTINUSE
: The current waveform device is in use. Wait until the device is free; then, try again to set the device for playback.

MCIERR_WAVE_SETOUTPUTUNSUITABLE
: The device you are using to playback a waveform cannot recognize the data format.

Comments Use the **MCI_OPEN** command to obtain the device ID specified by *wDeviceID*.

See Also **mciGetErrorString, mciSendString**

MCIERR_NEW_REQUIRES_ALIAS
You must specify an alias when using the "new" parameter.

MCIERR_NO_CLOSING_QUOTE
The string parameter is missing a closing double quotation mark, which you must supply.

MCIERR_NOTIFY_ON_AUTO_OPEN
You cannot use the "notify" flag with automatically opened devices.

MCIERR_PARAM_OVERFLOW
The output string was too large to fit in the return buffer. Increase the size of the buffer.

MCIERR_PARSER_INTERNAL
The device driver returned an invalid return type. Check with the device manufacturer about obtaining a new driver.

MCIERR_UNRECOGNIZED_KEYWORD
The driver cannot recognize the specified command parameter.

See Also **mciGetErrorString, mciSendCommand**

mciSetYieldProc

Syntax BOOL **mciSetYieldProc**(*wDeviceID, fpYieldProc, dwYieldData*)

This function sets the address of a callback procedure to be called periodically when an MCI device is completing a command specified with the WAIT flag.

Parameters UINT *wDeviceID*
Specifies the device ID of the MCI device to which the yield procedure is to be assigned.

YIELDPROC *fpYieldProc*
Specifies the callback procedure to be called when the given device is yielding. Specify a NULL value to disable any existing yield procedure.

DWORD *dwYieldData*
Specifies the data sent to the yield procedure when it is called for the given device.

Return Value Returns TRUE if successful. Returns FALSE for an invalid device ID.

Callback int CALLBACK **YieldProc**(*wDeviceID, dwData*)

YieldProc is a placeholder for the application-supplied function name. Export the actual name by including it in the EXPORTS statement in your module-definition file.

Comments The data buffer must be prepared with **midiInPrepareHeader** before it is passed to **midiInAddBuffer**. The **MIDIHDR** data structure and the data buffer pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**.

See Also **midiInPrepareHeader**

midiInClose

Syntax UINT **midiInClose**(*hMidiIn*)

This function closes the specified MIDI input device.

Parameters HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device. If the function is successful, the handle is no longer valid after this call.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MIDIERR_STILLPLAYING
There are still buffers in the queue.

Comments If there are input buffers that have been sent with **midiInAddBuffer** and haven't been returned to the application, the close operation will fail. Call **midiInReset** to mark all pending buffers as being done.

See Also **midiInOpen**, **midiInReset**

midiInGetErrorText

Syntax

UINT **midiInGetErrorText**(*wError*, *lpText*, *wSize*)

This function retrieves a text description of the error identified by the specified error number.

Parameters

UINT *wError*

Specifies the error number.

LPSTR *lpText*

Specifies a far pointer to the buffer to be filled with the text error description.

UINT *wSize*

Specifies the length of buffer pointed to by *lpText*.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADERRNUM

Specified error number is out of range.

Comments

If the text error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied, and the function returns zero. All error descriptions are less than MAXERRORLENGTH characters long.

midiInGetID

Syntax

UINT **midiInGetID**(*hMidiIn*, *lpwDeviceID*)

This function gets the device ID for a MIDI input device.

Parameters

HMIDIIN *hMidiIn*

Specifies the handle to the MIDI input device.

UINT FAR* *lpwDeviceID*

Specifies a pointer to the WORD-sized memory location to be filled with the device ID.

Return Value

Returns zero if successful. Otherwise, returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

The *hMidiIn* parameter specifies an invalid handle.

Parameters

LPHMIDIIN *lphMidiIn*

Specifies a far pointer to an HMIDIIN handle. This location is filled with a handle identifying the opened MIDI input device. Use the handle to identify the device when calling other MIDI input functions.

UINT *wDeviceID*

Identifies the MIDI input device to open. Specify a valid MIDI input device ID (see the following "Comments" section) or the following constant:

MIDI_MAPPER

MIDI mapper. The function will return an error if no MIDI mapper is installed.

DWORD *dwCallback*

Specifies the address of a fixed callback function or a handle to a window called with information about incoming MIDI messages.

DWORD *dwCallbackInstance*

Specifies user instance data passed to the callback function. This parameter is not used with window callbacks.

DWORD *dwFlags*

Specifies a callback flag for opening the device.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_ALLOCATED

Specified resource is already allocated.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

Callback

void CALLBACK **MidiInFunc**(*hMidiIn*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

MidiInFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the DLL's module definition file.

The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL, and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback should not make any system calls except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

See Also **midiInClose**

midiInPrepareHeader

Syntax UINT **midiInPrepareHeader**(*hMidiIn*, *lpMidiInHdr*, *wSize*)

This function prepares a buffer for MIDI input.

Parameters

HMIDIIN *hMidiIn*
Specifies a handle to the MIDI input device.

LPMIDIHDR *lpMidiInHdr*
Specifies a pointer to a **MIDIHDR** structure that identifies the buffer to be prepared.

UINT *wSize*
Specifies the size of the **MIDIHDR** structure.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MMSYSERR_NOMEM
Unable to allocate or lock memory.

Comments The **MIDIHDR** data structure and the data block pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header that has already been prepared has no effect, and the function returns zero.

See Also **midiInUnprepareHeader**

Comments This function resets the timestamps to zero; timestamp values for subsequently received messages are relative to the time this function was called.

All messages other than system-exclusive messages are sent directly to the client when received. System-exclusive messages are placed in the buffers supplied by **midiInAddBuffer**; if there are no buffers in the queue, the data is thrown away without notification to the client, and input continues.

Buffers are returned to the client when full, when a complete system-exclusive message has been received, or when **midiInReset** is called. The **dwBytesRecorded** field in the header will contain the actual length of data received.

Calling this function when input is already started has no effect, and the function returns zero.

See Also **midiInStop**, **midiInReset**

midiInStop

Syntax UINT **midiInStop**(*hMidiIn*)

This function terminates MIDI input on the specified MIDI input device.

Parameters HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments Current status (running status, parsing state, etc.) is maintained across calls to **midiInStop** and **midiInStart**. If there are any system exclusive message buffers in the queue, the current buffer is marked as done (the **dwBytesRecorded** field in the header will contain the actual length of data), but any empty buffers in the queue remain there. Calling this function when input is not started has no effect, and the function returns zero.

See Also **midiInStart**, **midiInReset**

midiOutCacheDrumPatches

Syntax

UINT **midiOutCacheDrumPatches**(*hMidiOut*, *wPatch*, *lpKeyArray*, *wFlags*)

This function requests that an internal MIDI synthesizer device preload a specified set of key-based percussion patches. Some synthesizers are not capable of keeping all percussion patches loaded simultaneously. Caching patches ensures specified patches are available.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the opened MIDI output device. This device should be an internal MIDI synthesizer.

UINT *wPatch*

Specifies which drum patch number should be used. To specify caching of the default drum patches, set this parameter to zero.

LPKEYARRAY *lpKeyArray*

Specifies a pointer to a **KEYARRAY** array indicating the key numbers of the specified percussion patches to be cached or uncached.

UINT *wFlags*

Specifies options for the cache operation. Only one of the following flags can be specified:

MIDI_CACHE_ALL

Cache all of the specified patches. If they can't all be cached, cache none, clear the **KEYARRAY** array, and return MMSYSERR_NOMEM.

MIDI_CACHE_BESTFIT

Cache all of the specified patches. If all patches can't be cached, cache as many patches as possible, change the **KEYARRAY** array to reflect which patches were cached, and return MMSYSERR_NOMEM.

MIDI_CACHE_QUERY

Change the **KEYARRAY** array to indicate which patches are currently cached.

MIDI_UNCACHE

Uncache the specified patches and clear the **KEYARRAY** array.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the opened MIDI output device. This device must be an internal MIDI synthesizer.

UINT *wBank*

Specifies which bank of patches should be used. To specify caching of the default patch bank, set this parameter to zero.

LPPATCHARRAY *lpPatchArray*

Specifies a pointer to a **PATCHARRAY** array indicating the patches to be cached or uncached.

UINT *wFlags*

Specifies options for the cache operation. Only one of the following flags can be specified:

MIDI_CACHE_ALL

Cache all of the specified patches. If they can't all be cached, cache none, clear the **PATCHARRAY** array, and return MMSYSERR_NOMEM.

MIDI_CACHE_BESTFIT

Cache all of the specified patches. If all patches can't be cached, cache as many patches as possible, change the **PATCHARRAY** array to reflect which patches were cached, and return MMSYSERR_NOMEM.

MIDI_CACHE_QUERY

Change the **PATCHARRAY** array to indicate which patches are currently cached.

MIDI_UNCACHE

Uncache the specified patches and clear the **PATCHARRAY** array.

Return Value

Returns zero if the function was successful. Otherwise, it returns one of the following error codes:

MMSYSERR_INVALHANDLE

The specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

The specified device does not support patch caching.

MMSYSERR_NOMEM

The device does not have enough memory to cache all of the requested patches.

midiOutGetDevCaps

Syntax

UINT **midiOutGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified MIDI output device to determine its capabilities.

Parameters

UINT *wDeviceID*

Identifies the MIDI output device to query. Specify a valid MIDI output device ID (see the following "Comments" section) or the following constant:

MIDI_MAPPER

MIDI mapper. The function will return an error if no MIDI mapper is installed.

LPMIDIOUTCAPS *lpCaps*

Specifies a far pointer to a **MIDIOUTCAPS** structure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the **MIDIOUTCAPS** structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use **midiOutGetNumDevs** to determine the number of MIDI output devices present in the system.

Only *wSize* bytes (or less) of information is copied to the location pointed to by *lpCaps*. If *wSize* is zero, nothing is copied, and the function returns zero.

See Also

midiOutGetNumDevs

midiOutGetNumDevs

Syntax UINT **midiOutGetNumDevs**()

This function retrieves the number of MIDI output devices present in the system.

Parameters None.

Return Value Returns the number of MIDI output devices present in the system.

See Also **midiOutGetDevCaps**

midiOutGetVolume

Syntax UINT **midiOutGetVolume**(*wDeviceID*, *lpdwVolume*)

This function returns the current volume setting of a MIDI output device.

Parameters UINT *wDeviceID*

Identifies the MIDI output device.

LPDWORD *lpdwVolume*

Specifies a far pointer to a location to be filled with the current volume setting. The low-order word of this location contains the left channel volume setting, and the high-order word contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of the specified location contains the mono volume level.

The full 16-bit setting(s) set with **midiOutSetVolume** is returned, regardless of whether the device supports the full 16 bits of volume-level control.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

MMSYSERR_NODRIVER

The driver was not installed.

Comments The data buffer must be prepared with **midiOutPrepareHeader** before it is passed to **midiOutLongMsg**. The **MIDIHDR** data structure and the data buffer pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. The MIDI output device driver determines whether the data is sent synchronously or asynchronously.

MIDI status is maintained across consecutive calls to **midiOutLongMsg** and **midiOutShortMsg**.

See Also **midiOutShortMsg**, **midiOutPrepareHeader**

midiOutMessage

Syntax DWORD **midiOutMessage**(*hMidiOut*, *msg*, *dwParam1*, *dwParam2*)

This function sends a message to a MIDI output device driver. Use it to send driver-specific messages that aren't supported by the MIDI APIs.

Parameters

HMIDIOUT *hMidiOut*
Specifies the handle to the audio device driver.

UINT *msg*
Specifies the message to send.

DWORD *dwParam1*
Specifies the first message parameter.

DWORD *dwParam2*
Specifies the second message parameter.

Return Value Returns the value returned by the audio device driver.

Comments Do not use this function to send standard messages to an audio device driver.

See Also **midiInMessage**

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are as follows:

MMSYSERR_BADDEVICEID
Specified device ID is out of range.

MMSYSERR_ALLOCATED
Specified resource is already allocated.

MMSYSERR_NOMEM
Unable to allocate or lock memory.

MIDIERR_NOMAP
There is no current MIDI map. This occurs only when opening the mapper.

MIDIERR_NODEVICE
A port in the current MIDI map doesn't exist. This occurs only when opening the mapper.

Callback

void CALLBACK **MidiOutFunc**(*hMidiOut*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

MidiOutFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the DLL's module-definition file.

Callback Parameters

HMIDIOUT *hMidiOut*
Specifies a handle to the MIDI device associated with the callback.

UINT *wMsg*
Specifies a MIDI output message.

DWORD *dwInstance*
Specifies the instance data supplied with **midiOutOpen**.

DWORD *dwParam1*
Specifies a parameter for the message.

DWORD *dwParam2*
Specifies a parameter for the message.

Comments The **MIDIHDR** data structure and the data block pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags and locked with **GlobalLock**. Preparing a header that has already been prepared has no effect, and the function returns zero.

See Also **midiOutUnprepareHeader**

midiOutReset

Syntax UINT **midiOutReset**(*hMidiOut*)

This function turns off all notes on all MIDI channels for the specified MIDI output device. If any long output buffers (from **midiOutLongMsg**) are pending, they are marked as done and returned to the application.

Parameters HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments To turn off all notes, a note-off message for each note for each channel is sent. In addition, the sustain controller is turned off for each channel.

See Also **midiOutLongMsg**, **midiOutClose**

midiOutSetVolume

Syntax UINT **midiOutSetVolume**(*wDeviceID*, *dwVolume*)

This function sets the volume of a MIDI output device.

Parameters UINT *wDeviceID*

Identifies the MIDI output device.

DWORD *dwVolume*

Specifies the new volume setting. The low-order word contains the left channel volume setting, and the high-order word contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of *dwVolume* specifies the volume level, and the high-order word is ignored.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MIDIERR_NOTREADY
The hardware is busy with other data.

Comments A system exclusive message can be started or completed using **midiOutShortMsg** by sending a start system exclusive message (0x000000F0) or an end of system exclusive message (0x000000F7); but, the system exclusive data bytes must be sent using **midiOutLongMsg**.

MIDI status is maintained across consecutive calls to **midiOutShortMsg** and **midiOutLongMsg**; but, a **midiOutShortMsg** message must contain all data bytes for a MIDI event.

midiOutShortMsg supports, as recommended usage, the status byte associated with each MIDI message; including the status byte with a MIDI message clarifies that message.

This function might not return until the message has been sent to the output device.

See Also **midiOutLongMsg**

midiOutUnprepareHeader

Syntax UINT **midiOutUnprepareHeader**(*hMidiOut*, *lpMidiOutHdr*, *wSize*)

This function cleans up the preparation performed by **midiOutPrepareHeader**. The **midiOutUnprepareHeader** function must be called after the device driver fills a data buffer and returns it to the application. You must call this function before freeing the data buffer.

Parameters HMIDIOUT *hMidiOut*
Specifies a handle to the MIDI output device.

LPMIDIHDR *lpMidiOutHdr*
Specifies a pointer to a **MIDIHDR** structure identifying the buffer to be cleaned up.

UINT *wSize*
Specifies the size of the **MIDIHDR** structure.

Return Value

The return value is zero if the operation is successful. Otherwise, the return value specifies an error code. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE
The contents of the buffer could not be written to disk.

MMIOERR_CANNOTREAD
An error occurred while re-filling the buffer.

MMIOERR_UNBUFFERED
The specified file is not opened for buffered I/O.

MMIOERR_CANNOTEXPAND
The specified memory file cannot be expanded, probably because the **adwInfo[0]** field was set to zero in the initial call to **mmioOpen**.

MMIOERR_OUTOFMEMORY
There was not enough memory to expand a memory file for further writing.

Comments

If the specified file is opened for writing or for both reading and writing, the I/O buffer will be flushed to disk before the next buffer is read. If the I/O buffer cannot be written to disk because the disk is full, then **mmioAdvance** will return MMIOERR_CANNOTWRITE.

If the specified file is only open for writing, the MMIO_WRITE flag must be specified.

If you have written to the I/O buffer, you must set the MMIO_DIRTY flag in the **dwFlags** field of the **MMIOINFO** structure before calling **mmioAdvance**. Otherwise, the buffer will not be written to disk.

If the end of file is reached, **mmioAdvance** will still return success, even though no more data can be read. Thus, to check for the end of the file, it is necessary to see if the **pchNext** and **pchEndRead** fields of the **MMIOINFO** structure are equal after calling **mmioAdvance**.

See Also

mmioGetInfo, **MMIOINFO**

mmioClose

Syntax UINT **mmioClose**(*hmmio*, *wFlags*)

This function closes a file opened with **mmioOpen**.

Parameters HMMIO *hmmio*

Specifies the file handle of the file to close.

UINT *wFlags*

Specifies options for the close operation.

MMIO_FHOPEN

If the file was opened by passing the MS-DOS file handle of an already-opened file to **mmioOpen**, then using this flag tells **mmioClose** to close the MMIO file handle, but not the MS-DOS file handle.

Return Value The return value is zero if the function is successful. Otherwise, the return value is an error code, either from **mmioFlush** or from the I/O procedure. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the buffer could not be written to disk.

See Also **mmioOpen**, **mmioFlush**

mmioCreateChunk

Syntax UINT **mmioCreateChunk**(*hmmio*, *lpck*, *wFlags*)

This function creates a chunk in a RIFF file opened with **mmioOpen**. The new chunk is created at the current file position. After the new chunk is created, the current file position is the beginning of the data portion of the new chunk.

mmioDescend

Syntax

UINT **mmioDescend**(*hmmio*, *lpck*, *lpckParent*, *wFlags*)

This function descends into a chunk of a RIFF file opened with **mmioOpen**. It can also search for a given chunk.

Parameters

HMMIO *hmmio*

Specifies the file handle of an open RIFF file.

LPMMCKINFO *lpck*

Specifies a far pointer to a caller-supplied **MMCKINFO** structure that **mmioDescend** fills with the following information:

- The **ckid** field is the chunk ID of the chunk.
- The **cksize** field is the size of the data portion of the chunk. The data size includes the form type or list type (if any), but does not include the 8-byte chunk header or the pad byte at the end of the data (if any).
- The **fccType** field is the form type if **ckid** is "RIFF", or the list type if **ckid** is "LIST". Otherwise, it is NULL.
- The **dwDataOffset** field is the file offset of the beginning of the data portion of the chunk. If the chunk is a "RIFF" chunk or a "LIST" chunk, then **dwDataOffset** is the offset of the form type or list type.
- The **dwFlags** contains other information about the chunk. Currently, this information is not used and is set to zero.

If the MMIO_FINDCHUNK, MMIO_FINDRIFF, or MMIO_FINDLIST flag is specified for *wFlags*, then the **MMCKINFO** structure is also used to pass parameters to **mmioDescend**:

- The **ckid** field specifies the four-character code of the chunk ID, form type, or list type to search for.

LPMMCKINFO *lpckParent*

Specifies a far pointer to an optional caller-supplied **MMCKINFO** structure identifying the parent of the chunk being searched for. A parent of a chunk is the enclosing chunk—only "RIFF" and "LIST" chunks can be parents. If *lpckParent* is not NULL, then **mmioDescend** assumes the **MMCKINFO** structure it refers to was filled when **mmioDescend** was called to descend into the parent chunk, and **mmioDescend** will only search for a chunk within the parent chunk. Set *lpckParent* to NULL if no parent chunk is being specified.

mmioFlush

Syntax

UINT **mmioFlush**(*hmmio*, *wFlags*)

This function writes the I/O buffer of a file to disk, if the I/O buffer has been written to.

Parameters

HMMIO *hmmio*

Specifies the file handle of a file opened with **mmioOpen**.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the buffer could not be written to disk.

Comments

Closing a file with **mmioClose** will automatically flush its buffer.

If there is insufficient disk space to write the buffer, **mmioFlush** will fail, even if the preceding **mmioWrite** calls were successful.

mmioFOURCC

Syntax

FOURCC **mmioFOURCC**(*ch0*, *ch1*, *ch2*, *ch3*)

This macro converts four characters to a four-character code.

Parameters

CHAR *ch0*

The first character of the four-character code.

CHAR *ch1*

The second character of the four-character code.

CHAR *ch2*

The third character of the four-character code.

CHAR *ch3*

The fourth character of the four-character code.

Return Value

The return value is the four-character code created from the given characters.

Comments

This macro does not check to see if the four character code follows any conventions regarding which characters to include in a four-character code.

See Also

mmioStringToFOURCC

mmioInstallIOProc

Syntax

LPMMIOPROC **mmioInstallIOProc**(*fccIOProc*, *pIOProc*, *dwFlags*)

This function installs or removes a custom I/O procedure. It will also locate an installed I/O procedure, given its corresponding four-character code.

Parameters

FOURCC *fccIOProc*

Specifies a four-character code identifying the I/O procedure to install, remove, or locate. All characters in this four-character code should be uppercase characters.

LPMMIOPROC *pIOProc*

Specifies the address of the I/O procedure to install. To remove or locate an I/O procedure, set this parameter to NULL.

DWORD *dwFlags*

Specifies one of the following flags indicating whether the I/O procedure is being installed, removed, or located:

MMIO_INSTALLPROC

Installs the specified I/O procedure. To allow other procedures to use the specified I/O procedure, also specify the MMIO_GLOBALPROC flag.

MMIO_REMOVEPROC

Removes the specified I/O procedure. When removing a global I/O procedure, only the task that registers a global I/O procedure can unregister that procedure.

MMIO_FINDPROC

Searches local, then global procedures for the specified I/O procedure.

MMIO_GLOBALPROC

Identifies the I/O procedure being installed as a global procedure.

Return Value

The return value is the address of the I/O procedure installed, removed, or located. If there is an error, the return value is NULL.

Callback

LONG FAR PASCAL **IOProc**(*lpmmioinfo*, *wMsg*, *lParam1*, *lParam2*)

IOProc is a placeholder for the application-supplied function name. The actual name must be exported by including it in a EXPORTS statement in the application's module-definitions file.

To share an I/O procedure among applications, each application can install and use local copies of the I/O procedure or one application can install a global copy of the I/O procedure for one or more applications to use. To use multiple, local copies of an I/O procedure among several applications, the I/O procedure must reside in a DLL called by each application using it. Each application using the shared I/O procedure must call **mmioInstallIOProc** to install the procedure (or call the DLL to install the procedure on behalf of the application). Each application must call **mmioInstallIOProc** to remove the I/O procedure before terminating.

If an application calls **mmioInstallIOProc** more than once to register the same local I/O procedure, then it must call **mmioInstallIOProc** to remove the procedure once for each time it installed the procedure.

mmioInstallIOProc will not prevent an application from installing two different I/O procedures with the same identifier, or installing an I/O procedure with one of the predefined four-character codes ("DOS ", "MEM ", or "BND "). The most recently installed procedure takes precedence and the most recently installed procedure is the first one to get removed.

To use a single copy of an I/O procedure among several applications, one application must install the I/O procedure as a global procedure. Then, other applications locate the global procedure before they use it. An application that installs a global I/O procedure can, without regard to other applications using the procedure, unregister that procedure at any time.

An application installs a global copy of an I/O procedure by calling **mmioInstallIOProc** with the flags MMIO_INSTALLPROC and MMIO_GLOBALPROC. Once an application globally installs a procedure, that application can use the global procedure. To unregister a procedure, the application that installed the procedure must call **mmioInstallIOProc**.

Other applications must locate an installed, global I/O procedure before using it. To locate a global procedure, an application calls **mmioInstallIOProc** with the flag MMIO_FINDPROC. Once an application locates the global procedure, it can call the procedure as needed. Applications that use, but do not install, a global I/O procedure, are exempt from actions to unregister that procedure.

See Also

mmioOpen

MMIO_WRITE

Opens the file for writing. You should not read from a file opened in this mode.

MMIO_READWRITE

Opens the file for both reading and writing.

MMIO_CREATE

Creates a new file. If the file already exists, it is truncated to zero length. For memory files, MMIO_CREATE indicates the end of the file is initially at the start of the buffer.

MMIO_DELETE

Deletes a file. If this flag is specified, *szFilename* should not be NULL. The return value will be TRUE (cast to HMMIO) if the file was deleted successfully, FALSE otherwise. Do not call **mmioClose** for a file that has been deleted. If this flag is specified, all other flags are ignored.

MMIO_PARSE

Creates a fully qualified filename from the path specified in *szFileName*. The fully qualified filename is placed back into *szFileName*. The return value will be TRUE (cast to HMMIO) if the qualification was successful, FALSE otherwise. The file is not opened, and the function does not return a valid MMIO file handle, so do not attempt to close the file. If this flag is specified, all other file opening flags are ignored.

MMIO_EXIST

Determines whether the specified file exists and creates a fully qualified filename from the path specified in *szFileName*. The fully qualified filename is placed back into *szFileName*. The return value will be TRUE (cast to HMMIO) if the qualification was successful and the file exists, FALSE otherwise. The file is not opened, and the function does not return a valid MMIO file handle, so do not attempt to close the file.

MMIO_ALLOCBUF

Opens a file for buffered I/O. To allocate a buffer larger or smaller than the default buffer size (8K), set the **cchBuffer** field of the **MMIOINFO** structure to the desired buffer size. If **cchBuffer** is zero, then the default buffer size is used. If you are providing your own I/O buffer, then the MMIO_ALLOCBUF flag should not be used.

MMIO_COMPAT

Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. **mmioOpen** fails if the file has been opened with any of the other sharing modes.

- To request that **mmioOpen** determine which I/O procedure to use to open the file based on the filename contained in *szFilename*, set both **fccIOProc** and **pIOProc** to NULL. This is the default behavior if no **MMIOINFO** structure is specified.
- To open a memory file using an internally allocated and managed buffer, set the **pchBuffer** field to NULL, **fccIOProc** to FOURCC_MEM, **cchBuffer** to the initial size of the buffer, and **adwInfo[0]** to the incremental expansion size of the buffer. This memory file will automatically be expanded in increments of **adwInfo[0]** bytes when necessary. Specify the MMIO_CREATE flag for the *dwOpenFlags* parameter to initially set the end of the file to be the beginning of the buffer.
- To open a memory file using a caller-supplied buffer, set the **pchBuffer** field to point to the memory buffer, **fccIOProc** to FOURCC_MEM, **cchBuffer** to the size of the buffer, and **adwInfo[0]** to the incremental expansion size of the buffer. The expansion size in **adwInfo[0]** should only be non-zero if **pchBuffer** is a pointer obtained by calling **GlobalAlloc** and **GlobalLock**, since **GlobalReAlloc** will be called to expand the buffer. In particular, if **pchBuffer** points to a local or global array, a block of memory in the local heap, or a block of memory allocated by **GlobalDosAlloc**, **adwInfo[0]** must be zero.

  Specify the MMIO_CREATE flag for the *dwOpenFlags* parameter to initially set the end of the file to be the beginning of the buffer; otherwise, the entire block of memory will be considered readable.
- To use a currently open MS-DOS file handle with MMIO, set the **fccIOProc** field to FOURCC_DOS, **pchBuffer** to NULL, and **adwInfo[0]** to the MS-DOS file handle. Note that offsets within the file will be relative to the beginning of the file, and will not depend on the MS-DOS file position at the time **mmioOpen** is called; the initial MMIO offset will be the same as the MS-DOS offset when **mmioOpen** is called. Later, to close the MMIO file handle without closing the MS-DOS file handle, pass the MMIO_FHOPEN flag to **mmioClose**.

You must call **mmioClose** to close a file opened with **mmioOpen**. Open files are not automatically closed when an application exits.

See Also **mmioClose**

mmioSeek

Syntax

LONG **mmioSeek**(*hmmio*, *lOffset*, *iOrigin*)

This function changes the current file position in a file opened with **mmioOpen**. The current file position is the location in the file where data is read or written.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file to seek in.

LONG *lOffset*

Specifies an offset to change the file position.

int *iOrigin*

Specifies how the offset specified by *lOffset* is interpreted. Contains one of the following flags:

SEEK_SET

Seeks to *lOffset* bytes from the beginning of the file.

SEEK_CUR

Seeks to *lOffset* bytes from the current file position.

SEEK_END

Seeks to *lOffset* bytes from the end of the file.

Return Value

The return value is the new file position in bytes, relative to the beginning of the file. If there is an error, the return value is –1.

Comments

Seeking to an invalid location in the file, such as past the end of the file, may not cause **mmioSeek** to return an error, but may cause subsequent I/O operations on the file to fail.

To locate the end of a file, call **mmioSeek** with *lOffset* set to zero and *iOrigin* set to SEEK_END.

LONG *cchBuffer*

Specifies the size of the caller-supplied buffer, or the size of the buffer for **mmioSetBuffer** to allocate.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. If an error occurs, the file handle remains valid. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the old buffer could not be written to disk, so the operation was aborted.

MMIOERR_OUTOFMEMORY

The new buffer could not be allocated, probably due to a lack of available memory.

Comments

To enable buffering using an internal buffer, set *pchBuffer* to NULL and *cchBuffer* to the desired buffer size.

To supply your own buffer, set *pchBuffer* to point to the buffer, and set *cchBuffer* to the size of the buffer.

To disable buffered I/O, set *pchBuffer* to NULL and *cchBuffer* to zero.

If buffered I/O is already enabled using an internal buffer, you can reallocate the buffer to a different size by setting *pchBuffer* to NULL and *cchBuffer* to the new buffer size. The contents of the buffer may be changed after resizing.

mmioSetInfo

Syntax

UINT **mmioSetInfo**(*hmmio*, *lpmmioinfo*, *wFlags*)

This function updates the information retrieved by **mmioGetInfo** about a file opened with **mmioOpen**. Use this function to terminate direct buffer access of a file opened for buffered I/O.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file.

LPMMIOINFO *lpmmioinfo*

Specifies a far pointer to an **MMIOINFO** structure filled with information with **mmioGetInfo**.

UINT *wFlags*

Is not used and should be set to zero.

Comments The current file position is incremented by the number of bytes written.

See Also **mmioRead**

mmsystemGetVersion

Syntax WORD **mmsystemGetVersion**()

This function returns the current version number of the multimedia system software.

0x0100 is the value returned with the Multimedia Extensions 1.0.
0x0101 is the value returned with version 3.1 of Windows.

Parameters None.

Return Value The return value specifies the major and minor version numbers of the multimedia system software. The high-order byte specifies the major version number. The low-order byte specifies the minor version number.

OutputDebugStr

Syntax void **OutputDebugStr**(*lpOutputString*)

This function sends a debugging message directly to the COM1 port or to a secondary monochrome display adapter. Because it bypasses MS-DOS, it can be called by low-level callback functions and other code at interrupt time.

Parameters LPCSTR *lpOutputString*

Specifies a far pointer to a null-terminated string.

Comments This function is available only in the debugging version of Windows. The DebugOutput keyname in the [mmsystem] section of SYSTEM.INI controls where the debugging information is sent. If DebugOutput is 0, all debug output is disabled. If DebugOutput is 1, debug output is sent to the COM1 port. If DebugOutput is 2, debug output is sent to a secondary monochrome display adapter.

To print the contents of a register, use the pound sign (#) followed by one of the following register designations: "ax", "bx", "cx", "dx", "si", "di", "bp", "sp", "al", "bl", "cl", "dl". For for systems that support the 80386 architecture, OutputDebugStr also supports the following registers: "fs", "gs", "edi", "esi", "eax", "ebx", "ecx", "edx".

For example, to print the stack pointer and accumulator registers, pass the following string to **OutputDebugStr**:

```
"SP=#sp\r\nAX=#ax\r\n"
```

Return Value Returns TRUE if the sound is played, otherwise returns FALSE.

Comments The sound must fit in available physical memory and be playable by an installed waveform audio device driver. The directories searched for sound files are, in order: the current directory; the Windows directory; the Windows system directory; the directories listed in the PATH environment variable; the list of directories mapped in a network. See the Windows **OpenFile** function for more information about the directory search order.

If you specify the SND_MEMORY flag, *lpszSoundName* must point to an in-memory image of a waveform sound. If the sound is stored as a resource, use **LoadResource** and **LockResource** to load and lock the resource and get a pointer to it. If the sound is not a resource, you must use **GlobalAlloc** with the GMEM_MOVEABLE and GMEM_SHARE flags set and then **GlobalLock** to allocate and lock memory for the sound.

timeBeginPeriod

Syntax UINT **timeBeginPeriod**(*wPeriod*)

This function sets the minimum (lowest number of milliseconds) timer resolution that an application or driver is going to use. Call this function immediately before starting to use timer-event services, and call **timeEndPeriod** immediately after finishing with the timer-event services.

Parameters UINT *wPeriod*

Specifies the minimum timer-event resolution that the application or driver will use.

Return Value Returns zero if successful. Returns TIMERR_NOCANDO if the specified *wPeriod* resolution value is out of range.

Comments For each call to **timeBeginPeriod**, you must call **timeEndPeriod** with a matching *wPeriod* value. An application or driver can make multiple calls to **timeBeginPeriod**, as long as each **timeBeginPeriod** call is matched with a **timeEndPeriod** call.

See Also **timeEndPeriod**, **timeSetEvent**

Parameters LPMMTIME *lpTime*
Specifies a far pointer to an **MMTIME** data structure.

UINT *wSize*
Specifies the size of the **MMTIME** structure.

Return Value Returns zero. The system time is returned in the **ms** field of the **MMTIME** structure.

Comments The time is always returned in milliseconds.

See Also **timeGetTime**

timeGetTime

Syntax DWORD **timeGetTime**()

This function retrieves the system time in milliseconds. The system time is the time elapsed since Windows was started.

Parameters None.

Return Value The return value is the system time in milliseconds.

Comments The only difference between this function and the **timeGetSystemTime** function is **timeGetSystemTime** uses the standard multimedia time structure **MMTIME** to return the system time. The **timeGetTime** function has less overhead than **timeGetSystemTime**.

See Also **timeGetSystemTime**

timeKillEvent

Syntax UINT **timeKillEvent**(*wTimerID*)

This functions destroys a specified timer callback event.

Parameters UINT *wTimerID*
Identifies the event to be destroyed.

Return Value Returns zero if successful. Returns TIMERR_NOCANDO if the specified timer event does not exist.

Comments The timer event ID specified by *wTimerID* must be an ID returned by **timeSetEvent**.

See Also **timeSetEvent**

Callback

void CALLBACK **TimeFunc**(*wTimerID*, *wMsg*, *dwUser*, *dw1*, *dw2*)

TimeFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in the EXPORTS statement of the module-definition file for the DLL.

Callback Parameters

UINT *wTimerID*
The ID of the timer event. This is the ID returned by **timeSetEvent**.

UINT *wMsg*
Not used.

DWORD *dwUser*
User instance data supplied to the *dwUser* parameter of **timeSetEvent**.

DWORD *dw1*
Not used.

DWORD *dw2*
Not used.

Comments

Using this function to generate a high-frequency periodic-delay event (with a period less than 10 milliseconds) can consume a significant portion of the system CPU bandwidth. Any call to **timeSetEvent** for a periodic-delay timer must be paired with a call to **timeKillEvent**.

The callback function must reside in a DLL. You don't have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL, and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

See Also

timeKillEvent, **timeBeginPeriod**, **timeEndPeriod**

WAVERR_STILLPLAYING
There are still buffers in the queue.

Comments If there are input buffers that have been sent with **waveInAddBuffer**, and haven't been returned to the application, the close operation will fail. Call **waveInReset** to mark all pending buffers as done.

See Also **waveInOpen, waveInReset**

waveInGetDevCaps

Syntax UINT **waveInGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified waveform input device to determine its capabilities.

Parameters UINT *wDeviceID*
Identifies the waveform input device to query. Use a valid waveform input device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER
Wave mapper. If no wave mapper is installed, the function returns an error number.

LPWAVEINCAPS *lpCaps*
Specifies a far pointer to a **WAVEINCAPS** structure. This structure is filled with information about the capabilities of the device.

UINT *wSize*
Specifies the size of the **WAVEINCAPS** structure.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID
Specified device ID is out of range.

MMSYSERR_NODRIVER
The driver was not installed.

Comments The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use **waveInGetNumDevs** to determine the number of waveform input devices present in the system.

Only *wSize* bytes (or less) of information is copied to the location pointed to by *lpCaps*. If *wSize* is zero, nothing is copied, and the function returns zero.

See Also **waveInGetNumDevs**

waveInGetNumDevs

Syntax UINT **waveInGetNumDevs**()

This function returns the number of waveform input devices.

Parameters None.

Return Value Returns the number of waveform input devices present in the system.

See Also **waveInGetDevCaps**

waveInGetPosition

Syntax UINT **waveInGetPosition**(*hWaveIn*, *lpInfo*, *wSize*)

This function retrieves the current input position of the specified waveform input device.

Parameters

HWAVEIN *hWaveIn*
Specifies a handle to the waveform input device.

LPMMTIME *lpInfo*
Specifies a far pointer to an **MMTIME** structure.

UINT *wSize*
Specifies the size of the **MMTIME** structure.

Return Value Returns zero if the function was successful. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

Comments Before calling **waveInGetPosition**, set the **wType** field of the **MMTIME** structure to indicate the time format that you desire. After calling **waveInGetPosition**, be sure to check the **wType** field to determine if the desired time format is supported. If the desired format is not supported, **wType** will specify an alternative format.

The position is set to zero when the device is opened or reset.

UINT *wDeviceID*

Identifies the waveform input device to open. Use a valid waveform input device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER

Wave mapper. If no wave mapper is installed, the system selects a waveform input device capable of recording in the given format.

LPWAVEFORMAT *lpFormat*

Specifies a pointer to a **WAVEFORMAT** data structure that identifies the desired format for recording waveform data.

DWORD *dwCallback*

Specifies the address of a callback function or a handle to a window called during waveform recording to process messages related to the progress of recording.

DWORD *dwCallbackInstance*

Specifies user instance data passed to the callback. This parameter is not used with window callbacks.

DWORD *dwFlags*

Specifies flags for opening the device.

WAVE_FORMAT_QUERY

If this flag is specified, the device will be queried to determine if it supports the given format but will not actually be opened.

WAVE_ALLOWSYNC

Allows a synchronous (blocking) waveform driver to be opened. If this flag is not set while opening a synchronous driver, the open will fail.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

If a window is chosen to receive callback information, the following messages are sent to the window procedure function to indicate the progress of waveform input: **MM_WIM_OPEN**, **MM_WIM_CLOSE**, **MM_WIM_DATA**.

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of waveform input: **WIM_OPEN**, **WIM_CLOSE**, **WIM_DATA**. The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

See Also **waveInClose**

waveInPrepareHeader

Syntax

UINT **waveInPrepareHeader**(*hWaveIn*, *lpWaveInHdr*, *wSize*)

This function prepares a buffer for waveform input.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

LPWAVEHDR *lpWaveInHdr*

Specifies a pointer to a **WAVEHDR** structure that identifies the buffer to be prepared.

UINT *wSize*

Specifies the size of the **WAVEHDR** structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

Comments

The **WAVEHDR** data structure and the data block pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header that has already been prepared will have no effect, and the function will return zero.

See Also **waveInUnprepareHeader**

waveInStop

Syntax

UINT **waveInStop**(*hWaveIn*)

This function stops waveform input.

Parameters

HWAVEIN *hWaveIn*
Specifies a handle to the waveform input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

Comments

If there are any buffers in the queue, the current buffer will be marked as done (the **dwBytesRecorded** field in the header will contain the actual length of data), but any empty buffers in the queue will remain there. Calling this function when input is not started has no effect, and the function returns zero.

See Also

waveInStart, **waveInReset**

waveInUnprepareHeader

Syntax

UINT **waveInUnprepareHeader**(*hWaveIn*, *lpWaveInHdr*, *wSize*)

This function cleans up the preparation performed by **waveInPrepareHeader**. The function must be called after the device driver fills a data buffer and returns it to the application. You must call this function before freeing the data buffer.

Parameters

HWAVEIN *hWaveIn*
Specifies a handle to the waveform input device.

LPWAVEHDR *lpWaveInHdr*
Specifies a pointer to a **WAVEHDR** structure identifying the data buffer to be cleaned up.

UINT *wSize*
Specifies the size of the **WAVEHDR** structure.

waveOutClose

Syntax

UINT **waveOutClose**(*hWaveOut*)

This function closes the specified waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device. If the function is successful, the handle is no longer valid after this call.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_STILLPLAYING

There are still buffers in the queue.

Comments

If the device is still playing a waveform, the close operation will fail. Use **waveOutReset** to terminate waveform playback before calling **waveOutClose**.

See Also

waveOutOpen, **waveOutReset**

waveOutGetDevCaps

Syntax

UINT **waveOutGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified waveform device to determine its capabilities.

Parameters

UINT *wDeviceID*

Identifies the waveform output device to query. Use a valid waveform output device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER

Wave mapper. If no wave mapper is installed, the function returns an error number.

LPWAVEOUTCAPS *lpCaps*

Specifies a far pointer to a **WAVEOUTCAPS** structure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the **WAVEOUTCAPS** structure.

Comments If the text error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied, and the function returns zero. All error descriptions are less than MAXERRORLENGTH characters long.

waveOutGetID

Syntax UINT **waveOutGetID**(*hWaveOut, lpwDeviceID*)

This function gets the device ID for a waveform output device.

Parameters HWAVEOUT *hWaveOut*

Specifies the handle to the waveform output device.

UINT FAR* *lpwDeviceID*

Specifies a pointer to the WORD-sized memory location to be filled with the device ID.

Return Value Returns zero if successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

The *hWaveOut* parameter specifies an invalid handle.

waveOutGetNumDevs

Syntax UINT **waveOutGetNumDevs**()

This function retrieves the number of waveform output devices present in the system.

Parameters None.

Return Value Returns the number of waveform output devices present in the system.

See Also **waveOutGetDevCaps**

waveOutGetPlaybackRate

Syntax

UINT **waveOutGetPlaybackRate**(*hWaveOut*, *lpdwRate*)

This function queries the current playback rate setting of a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPDWORD *lpdwRate*

Specifies a far pointer to a location to be filled with the current playback rate. The playback rate setting is a multiplier indicating the current change in playback rate from the original authored setting. The playback rate multiplier must be a positive value.

The rate is specified as a fixed-point value. The high-order word of the DWORD location contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction is expressed as a WORD in which a value of 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no playback rate change), and a value of 0x000F8000 specifies a multiplier of 15.5.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

Comments

Changing the playback rate does not change the sample rate but does change the playback time.

Not all devices support playback rate changes. To determine whether a device supports playback rate changes, use the WAVECAPS_PLAYBACKRATE flag to test the **dwSupport** field of the **WAVEOUTCAPS** structure (filled by **waveOutGetDevCaps**).

See Also

waveOutSetPlaybackRate, **waveOutSetPitch**, **waveOutGetPitch**

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED
Function isn't supported.

MMSYSERR_NODRIVER
The driver was not installed.

Comments Not all devices support volume changes. To determine whether the device supports volume control, use the WAVECAPS_VOLUME flag to test the **dwSupport** field of the **WAVEOUTCAPS** structure (filled by **waveOutGetDevCaps**).

To determine whether the device supports volume control on both the left and right channels, use the WAVECAPS_VOLUME flag to test the **dwSupport** field of the **WAVEOUTCAPS** structure (filled by **waveOutGetDevCaps**).

See Also **waveOutSetVolume**

waveOutMessage

Syntax DWORD **waveOutMessage**(*hWaveOut*, *msg*, *dw1*, *dw2*)

This function sends a message to a waveform output device driver. Use it to send driver-specific messages that aren't supported by the waveform APIs.

Parameters HWAVEOUT *hWaveOut*
Specifies the handle to the audio device driver.

UINT *msg*
Specifies the message to send.

DWORD *dw1*
Specifies the first message parameter.

DWORD *dw2*
Specifies the second message parameter.

Return Value Returns the value returned by the audio device driver.

Comments Do not use this function to send standard messages to an audio device driver.

See Also **waveInMessage**

DWORD *dwFlags*

Specifies flags for opening the device.

WAVE_FORMAT_QUERY

If this flag is specified, the device is be queried to determine if it supports the given format but is not actually opened.

WAVE_ALLOWSYNC

Allows a synchronous (blocking) waveform driver to be opened. If this flag is not set while opening a synchronous driver, the open will fail.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_ALLOCATED

Specified resource is already allocated.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

WAVERR_BADFORMAT

Attempted to open with an unsupported wave format.

WAVERR_SYNC

Attempted to open a synchronous driver without specifying the WAVE_ALLOWSYNC flag.

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of waveform output: **WOM_OPEN**, **WOM_CLOSE**, **WOM_DONE**. The callback function must reside in a DLL. You don't have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

See Also **waveOutClose**

waveOutPause

Syntax UINT **waveOutPause**(*hWaveOut*)

This function pauses playback on a specified waveform output device. The current playback position is saved. Use **waveOutRestart** to resume playback from the current playback position.

Parameters HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments Calling this function when the output is already paused has no effect, and the function returns zero.

See Also **waveOutRestart**, **waveOutBreakLoop**

waveOutReset

Syntax

UINT **waveOutReset**(*hWaveOut*)

This function stops playback on a given waveform output device and resets the current position to 0. All pending playback buffers are marked as done and returned to the application.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

See Also

waveOutWrite, **waveOutClose**

waveOutRestart

Syntax

UINT **waveOutRestart**(*hWaveOut*)

This function restarts a paused waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Calling this function when the output is not paused has no effect, and the function returns zero.

See Also

waveOutPause, **waveOutBreakLoop**

waveOutSetPlaybackRate

Syntax

UINT **waveOutSetPlaybackRate**(*hWaveOut*, *dwRate*)

This function sets the playback rate of a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

DWORD *dwRate*

Specifies the new playback rate setting. The playback rate setting is a multiplier indicating the current change in playback rate from the original authored setting. The playback rate multiplier must be a positive value.

The rate is specified as a fixed-point value. The high-order word contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction is expressed as a WORD in which a value of 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no playback rate change), and a value of 0x000F8000 specifies a multiplier of 15.5.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

Comments

Changing the playback rate does not change the sample rate but does change the playback time.

Not all devices support playback rate changes. To determine whether a device supports playback rate changes, use the WAVECAPS_PLAYBACKRATE flag to test the **dwSupport** field of the **WAVEOUTCAPS** structure (filled by **waveOutGetDevCaps**).

See Also

waveOutGetPlaybackRate, **waveOutSetPitch**, **waveOutGetPitch**

waveOutUnprepareHeader

Syntax

UINT **waveOutUnprepareHeader**(*hWaveOut*, *lpWaveOutHdr*, *wSize*)

This function cleans up the preparation performed by **waveOutPrepareHeader**. The function must be called after the device driver is finished with a data block. You must call this function before freeing the data buffer.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPWAVEHDR *lpWaveOutHdr*

Specifies a pointer to a **WAVEHDR** structure identifying the data block to be cleaned up.

UINT *wSize*

Specifies the size of the **WAVEHDR** structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_STILLPLAYING

lpWaveOutHdr is still in the queue.

Comments

This function is the complementary function to **waveOutPrepareHeader**. You must call this function before freeing the data buffer with **GlobalFree**. After passing a buffer to the device driver with **waveOutWrite**, you must wait until the driver is finished with the buffer before calling **waveOutUnprepareHeader**.

Unpreparing a buffer that has not been prepared has no effect, and the function returns zero.

See Also

waveOutPrepareHeader

Chapter 4

Message Overview

This chapter gives an overview of the multimedia messages in Windows. The messages are organized into the following categories, some of which contain smaller groups of related messages:

- Audio Messages
- Media Control Interface Messages
- Joystick Messages
- File I/O Messages

For detailed information on any of the messages listed in this chapter, see Chapter 5, "Message Directory." Chapter 5 is an alphabetical listing of the messages in the Multimedia extensions to Windows.

About the Multimedia Messages

The multimedia messages fall into two broad categories:

- Messages sent to windows. These are processed by window functions and are similar to the messages defined in the WINDOWS.H header file. Multimedia messages sent to windows all have an MM_ prefix.
- Messages specific to a callback function or message-based API. These include Media Control Interface command messages, which an application sends to communicate with MCI, as well as messages sent to callback functions.

This chapter describes each message in detail.

Waveform Input Messages

Waveform input messages are sent by audio device drivers to an application to inform the application about the status of waveform input operations. By specifying flags with the **waveInOpen** function, applications can route messages either to a window or to a low-level callback function. Use the following messages to manage waveform audio recording:

MM_WIM_CLOSE

Sent to a window when a waveform input device is closed.

MM_WIM_DATA

Sent to a window when an input data buffer is full and is being returned to the application.

MM_WIM_OPEN

Sent to a window when a waveform input device is opened.

WIM_CLOSE

Sent to a low-level callback function when a waveform input device is closed.

WIM_DATA

Sent to a low-level callback function when an input data buffer is full and is being returned to the application.

WIM_OPEN

Sent to a low-level callback function when a waveform input device is opened.

MM_MIM_ERROR

Sent to a window when an invalid MIDI message is received by the device.

MM_MIM_LONGERROR

Sent to a window when an invalid MIDI system-exclusive message is received by the device.

MM_MIM_LONGDATA

Sent to a window when a MIDI system-exclusive data buffer is filled and is being returned to the application.

MM_MIM_OPEN

Sent to a low-level callback function when a MIDI input device is opened.

MIM_OPEN

Sent to a window when a MIDI input device is opened.

MIM_CLOSE

Sent to a low-level callback function when a MIDI input device is closed.

MIM_DATA

Sent to a low-level callback function when a MIDI message is received by the device. The parameters to this message include a time stamp specifying the time that the MIDI message was received.

MIM_ERROR

Sent to a low-level callback function when an invalid MIDI message is received by the device.

MIM_LONGERROR

Sent to a low-level callback function when an invalid MIDI system-exclusive message is received by the device.

MIM_LONGDATA

Sent to a low-level callback function when a MIDI system-exclusive message is received by the device. The parameters to this message include a time stamp specifying the time that the MIDI message was received.

Playing and Recording Multimedia Data

The following command messages control the playback and recording of multimedia data:

MCI_PAUSE
Sent by an application to pause a device.

MCI_PLAY
Sent by an application to start a device playing.

MCI_RECORD
Sent by an application to start recording with a device.

MCI_RESUME
Sent by an application to resume playback or recording after a pause.

MCI_STEP
Sent by an application to step a device one or more frames.

MCI_STOP
Sent by an application to stop a device from playing or recording.

Getting Device Information

The following command messages return information about devices:

MCI_GETDEVCAPS
Sent by an application to obtain information about device capabilities.

MCI_INFO
Sent by an application to obtain information about a device.

MCI_STATUS
Sent by an application to obtain status information about a device.

MCI_SYSINFO
Sent by an application to obtain system-related information about a device.

MCI_LOAD
Sent by an application to load a file.

MCI_PASTE
Sent by an application to paste data from the Clipboard to the MCI element.

MCI_SAVE
Sent by an application to save the current file.

Controlling Video Images

The following command messages control the presentation of video images:

MCI_FREEZE
Sent by an application to stop capture.

MCI_PUT
Sent by an application to define a source or destination clipping rectangle.

MCI_REALIZE
Sent by an application to tell a graphic device to realize its palette.

MCI_UNFREEZE
Sent by an application to restore capture.

MCI_UPDATE
Sent by an application to tell a graphic device to update or paint the current frame.

MCI_WHERE
Sent by an application to determine the extent of a clipping rectangle.

MCI_WINDOW
Sent by an application to specify a window and the characteristics of the window for a graphic device to use for its display.

MM_JOY2BUTTONUP
Sent to a window that has captured joystick 2 when a button has been released.

MM_JOY2MOVE
Sent to a window that has captured joystick 2 when the joystick position has changed.

MM_JOY2ZMOVE
Sent to a window that has captured joystick 2 when the joystick z-axis position has changed.

File I/O Messages

File I/O messages are sent to custom I/O procedures to request I/O operations on a file. I/O procedures must respond to all of the following messages:

MMIOM_CLOSE
Sent to an I/O procedure to request that a file be closed.

MMIOM_OPEN
Sent to an I/O procedure to request that a file be opened.

MMIOM_READ
Sent to an I/O procedure to request that data be read from a file.

MMIOM_RENAME
Sent to an I/O procedure to request that a file be renamed.

MMIOM_SEEK
Sent to an I/O procedure to request that the current position for reading and writing be changed.

MMIOM_WRITE
Sent to an I/O procedure to request that data be written to a file.

MMIOM_WRITEFLUSH
Sent to an I/O procedure to request that an I/O buffer be flushed to disk.

Chapter 5

Message Directory

This chapter contains an alphabetical list of the Windows multimedia messages. For information about standard Windows messages, see the *Microsoft Windows Programmer's Reference, Volume 3: Messages, Structures, Macros.*

For each message, this chapter lists the following items:

- The purpose of the message
- A description of the message parameters
- A description of return values
- Optional comments on using the message
- Optional cross references to other messages, functions, and data structures

Extensions to MCI Command Messages

MCI command messages can have extensions for specific MCI device types. The extensions include additional flags for parameters and are identified by one of the following headings in the "Parameters" section of the entry for the message:

Heading	Description
Animation Extensions	Extensions for the *animation* device type
Audio CD Extensions	Extensions for the *cdaudio* device type
Sequencer Extensions	Extensions for the *sequencer* device type
Videodisc Extensions	Extensions for the *videodisc* player devices
Video Overlay Extensions	Extensions for the *overlay* device type
Wave Audio Extensions	Extensions for the *waveaudio* device type

MCI_BREAK

This MCI command message sets a break key for an MCI device. MCI supports this message directly rather than passing it to the device.

Parameters

DWORD *dwParam1*

The following flags apply to all devices:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpBreak*.

MCI_WAIT

Specifies that the break operation should finish before MCI returns control to the application.

MCI_BREAK_KEY

Indicates the **nVirtKey** field of the data structure identified by *lpBreak* specifies the virtual key code used for the break key. By default, MCI assigns CTRL+BREAK as the break key. This flag is required if MCI_BREAK_OFF is not specified.

MCI_BREAK_HWND

Indicates the **hwndBreak** field of the data structure identified by *lpBreak* contains a window handle which must be the current window in order to enable break detection for that MCI device. This is usually the application's main window. If omitted, MCI does not check the window handle of the current window.

MCI_BREAK_OFF

Used to disable any existing break key for the indicated device.

DWORD *dwParam2*

Specifies a far pointer to the **MCI_BREAK_PARMS** data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

You might have to press the break key multiple times to interrupt a wait operation. Pressing the break key after a device wait is canceled can send the break to an application. If an application has an action defined for the virtual key code, then it can inadvertently respond to the break. For example, an application using VK_CANCEL for an accelerator key can respond to the default CTRL+BREAK key if it is pressed after a wait is canceled.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_COPY**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpCopy*.

MCI_WAIT

Specifies that the copy should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpCopy*

Specifies a far pointer to an **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_CUT, MCI_DELETE, MCI_PASTE

MCI_CUE

This MCI command message cues a device so that playback or recording begins with minimum delay. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_CUE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the cue operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MCI_DELETE

This MCI command message removes data from the MCI element. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_DELETE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpDelete*.

MCI_WAIT

Specifies that the delete operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpCut*

Specifies a far pointer to an **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Wave Audio Extensions

The following additional command options are used with the "waveaudio" device type:

DWORD *dwFlags*

The following extensions apply to the "waveaudio" device type:

MCI_FROM

Specifies that a beginning position is included in the **dwFrom** field of the data structure identified by *lpDelete*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

MCI_TO

Specifies that an ending position is included in the **dwTo** field of the data structure identified by *lpDelete*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

LPMCI_WAVE_DELETE_PARMS *lpDelete*

Specifies a far pointer to an **MCI_WAVE_DELETE_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MCI_FREEZE (VIDEO OVERLAY)

This MCI command message freezes motion on the display. This command is part of the video overlay command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_FREEZE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpFreeze*.

MCI_WAIT

Specifies that the freeze operation should finish before MCI returns control to the application.

MCI_OVLY_RECT

Specifies that the **rc** field of the data structure identified by *lpFreeze* contains a valid rectangle. If this flag is not specified, the device driver will freeze the entire frame.

LPMCI_OVLY_RECT_PARMS *lpFreeze*

Specifies a far pointer to a **MCI_OVLY_RECT_PARMS** data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_UNFREEZE

MCI_GETDEVCAPS_COMPOUND_DEVICE

The **dwReturn** field is set to TRUE if the device uses device elements; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_DEVICE_TYPE

The **dwReturn** field is set to one of the following values indicating the device type:

- MCI_DEVTYPE_ANIMATION
- MCI_DEVTYPE_CD_AUDIO
- MCI_DEVTYPE_DAT
- MCI_DEVTYPE_DIGITAL_VIDEO
- MCI_DEVTYPE_OTHER
- MCI_DEVTYPE_OVERLAY
- MCI_DEVTYPE_SCANNER
- MCI_DEVTYPE_SEQUENCER
- MCI_DEVTYPE_VIDEODISC
- MCI_DEVTYPE_VIDEOTAPE
- MCI_DEVTYPE_WAVEFORM_AUDIO

MCI_GETDEVCAPS_HAS_AUDIO

The **dwReturn** field is set to TRUE if the device has audio output; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_HAS_VIDEO

The **dwReturn** field is set to TRUE if the device has video output; otherwise, it is set to FALSE.

For example, the field is set to TRUE for devices that support the animation or videodisc command set.

MCI_GETDEVCAPS_USES_FILES

The **dwReturn** field is set to TRUE if the device requires a filename as its element name; otherwise, it is set to FALSE.

Only compound devices use files.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*

Specifies a far pointer to the **MCI_GETDEVCAPS_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Videodisc Extensions

The following additional command options are used with the "videodisc" device type:

DWORD *dwFlags*

The following extensions apply to the videodisc "device" type:

MCI_GETDEVCAPS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lpCapsParms* contains a constant specifying which device capability to obtain. The following additional device-capability constants are defined for videodisc devices and specify which value to return in the **dwReturn** field of the data structure:

MCI_VD_GETDEVCAPS_CAN_REVERSE

The **dwReturn** field is set to TRUE if the videodisc player can play in reverse; otherwise, it is set to FALSE.

Some players can play CLV discs in reverse as well as CAV discs.

MCI_VD_GETDEVCAPS_FAST_RATE

The **dwReturn** field is set to the standard fast play rate in frames per second.

MCI_VD_GETDEVCAPS_NORMAL_RATE

The **dwReturn** field is set to the normal play rate in frames per second.

MCI_VD_GETDEVCAPS_SLOW_RATE

The **dwReturn** field is set to the standard slow play rate in frames per second.

MCI_VD_GETDEVCAPS_CLV

Indicates the information requested applies to CLV format discs. By default, the capabilities apply to the current disc.

MCI_VD_GETDEVCAPS_CAV

Indicates the information requested applies to CAV format discs. By default, the capabilities apply to the current disc.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*

Specifies a far pointer to the **MCI_GETDEVCAPS_PARMS** data structure.

MCI_WAVE_GETDEVCAPS_OUTPUT
The **dwReturn** field is set to the total number of waveform output (playback) devices.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*
Specifies a far pointer to the **MCI_GETDEVCAPS_PARMS** data structure.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_INFO

This MCI command message obtains string information from a device. All devices respond to this message. The parameters and flags available for this message depend on the selected device. Information is returned in the **lpstrReturn** field of the data structure identified by *lpInfo*. The **dwRetSize** field specifies the buffer length for the return data.

Parameters DWORD *dwFlags*
The following standard and command-specific flags apply to all devices:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpInfo*.

MCI_WAIT
Specifies that the query operation should finish before MCI returns control to the application.

MCI_INFO_PRODUCT
Obtains a description of the hardware associated with a device. Devices should supply a description that identifies both the driver and the hardware used.

LPMCI_INFO_PARMS *lpInfo*
Specifies a far pointer to the **MCI_INFO_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MCI_WAVE_INPUT
Obtains the product name of the current input.

MCI_WAVE_OUTPUT
Obtains the product name of the current output.

LPMCI_INFO_PARMS *lpInfo*
Specifies a far pointer to the **MCI_INFO_PARMS** data structure.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_LOAD

This MCI command message loads a file. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters DWORD *dwFlags*
The following flags apply to all devices supporting **MCI_LOAD**:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpLoad.*

MCI_WAIT
Specifies that the load operation should finish before MCI returns control to the application.

MCI_LOAD_FILE
Indicates the **lpfilename** field of the data structure identified by *lpLoad* contains a pointer to a buffer containing the filename.

LPMCI_LOAD_PARMS *lpLoad*
Specifies a far pointer to the **MCI_LOAD_PARMS** data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

MCI_OPEN_SHAREABLE

Specifies that the device or device element should be opened as shareable.

MCI_OPEN_TYPE

Specifies that a device type name or constant is included in the **lpstrDeviceType** field of the data structure identified by *lpOpen*.

MCI_OPEN_TYPE_ID

Specifies that the low-order word of the **lpstrDeviceType** field of the associated data structure contains a standard MCI device type ID and the high-order word optionally contains the ordinal index for the device. Use this flag with the MCI_OPEN_TYPE flag.

LPMCI_OPEN_PARMS *lpOpen*

Specifies a far pointer to the **MCI_OPEN_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Flags for Compound Devices

The following additional command options are used with compound devices:

DWORD *dwFlags*

The following additional flags apply to compound devices:

MCI_OPEN_ELEMENT

Specifies that an element name is included in the **lpstrElementName** field of the data structure identified by *lpOpen*.

MCI_OPEN_ELEMENT_ID

Specifies that the **lpstrElementName** field of the data structure identified by *lpOpen* is interpreted as a DWORD and has meaning internal to the device. Use this flag with the MCI_OPEN_ELEMENT flag.

LPMCI_OPEN_PARMS *lpOpen*

Specifies a far pointer to the **MCI_OPEN_PARMS** data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Video Overlay Extensions

The following additional command options are used with the "overlay" device type:

DWORD *dwFlags*

The following flags apply to video overlay devices:

MCI_OVLY_OPEN_PARENT

Indicates the parent window handle is specified in the **hWndParent** field of the data structure identified by *lpOpen*.

MCI_OVLY_OPEN_WS

Indicates a window style is specified in the **dwStyle** field of the data structure identified by *lpOpen*. The **dwStyle** value specifies the style of the window that the driver will create and display if the application does not provide one. The style parameter takes an integer that defines the window style. These constants are the same as those in WINDOWS.H (for example, WS_CHILD, WS_OVERLAPPEDWINDOW, or WS_POPUP).

LPMCI_OVLY_OPEN_PARMS *lpOpen*

Specifies a far pointer to the **MCI_OVLY_OPEN_PARMS** data structure.

Waveform Audio Extensions

The following additional command options are used with the "waveaudio" device type:

DWORD *dwFlags*

The following flags apply to waveform audio devices:

MCI_WAVE_OPEN_BUFFER

Indicates a buffer length is specified in the **dwBufferSeconds** field of the data structure identified by *lpOpen*.

LPMCI_WAVE_OPEN_PARMS *lpOpen*

Specifies a far pointer to the **MCI_WAVE_OPEN_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MCI_PASTE

This MCI command message pastes data from the Clipboard into a device element.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_PASTE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpPaste*.

MCI_WAIT

Specifies that the device should complete the operation before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpPaste*

Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_CUT, MCI_COPY, MCI_DELETE

MCI_PAUSE

This MCI command message pauses the current action.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_PAUSE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the device should be paused before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

LPMCI_PLAY_PARMS *lpPlay*

Specifies a far pointer to an **MCI_PLAY_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

The following additional command options are used with the "animation" device type:

DWORD *dwFlags*

The following additional flags apply to animation devices:

MCI_ANIM_PLAY_FAST

Specifies to play fast.

MCI_ANIM_PLAY_REVERSE

Specifies to play in reverse.

MCI_ANIM_PLAY_SCAN

Specifies to scan quickly.

MCI_ANIM_PLAY_SLOW

Specifies to play slowly.

MCI_ANIM_PLAY_SPEED

Specifies that the play speed is included in the **dwSpeed** field in the data structure identified by *lpPlay*.

LPMCI_ANIM_PLAY_PARMS *lpPlay*

Specifies a far pointer to an **MCI_ANIM_PLAY_PARMS** data structure.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_PUT**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpDest*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDest*

Specifies a far pointer to an **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

The following additional command options are used with the "animation" device type:

DWORD *dwFlags*

The following additional flags apply to animation devices supporting **MCI_PUT**:

MCI_ANIM_RECT

Specifies that the **rc** field of the data structure identified by *lpDest* contains a valid rectangle. If this flag is not specified, the default rectangle matches the coordinates of the image or window being clipped.

MCI_ANIM_PUT_DESTINATION

Indicates the rectangle defined for MCI_ANIM_RECT specifies the area of the client window used to display an image. The rectangle contains the offset and visible extent of the image relative to the window origin. If the frame is being stretched, the source is stretched to the destination rectangle.

MCI_ANIM_PUT_SOURCE

Indicates the rectangle defined for MCI_ANIM_RECT specifies a clipping rectangle for the animation image. The rectangle contains the offset and extent of the image relative to the image origin.

LPMCI_ANIM_RECT_PARMS *lpDest*

Specifies a far pointer to a **MCI_ANIM_RECT_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MCI_REALIZE (ANIMATION)

This MCI command message tells a graphic device to realize its palette into a device context. This is part of the animation command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_REALIZE**:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpRealize*.

MCI_WAIT
: Specifies that the palette should be realized before MCI returns control to the application.

MCI_ANIM_REALIZE_BKGD
: If this flag is set, the palette is realized as a background palette.

MCI_ANIM_REALIZE_NORM
: If this flag is set, the palette is realized normally. This is the default action.

LPMCI_GENERIC_PARMS *lpRealize*
: Specifies a far pointer to a **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command is supported by devices that return true to the MCI_GETDEVCAPS_PALETTES query.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION for this command.

Comments This command is supported by devices that return TRUE to the MCI_GETDEVCAPS_CAN_RECORD query.

See Also **MCI_CUE, MCI_PAUSE, MCI_PLAY, MCI_RESUME, MCI_SEEK**

MCI_RESUME

This MCI command message resumes a paused device. Support of this message by a device is optional.

Parameters DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_RESUME**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the device should resume before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

Comments This command resumes playing and recording without changing the stop position set with **MCI_PLAY** or **MCI_RECORD**.

See Also **MCI_STOP, MCI_PLAY, MCI_RECORD**

Return Value Returns zero if successful. Otherwise, it returns an MCI error code. MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION.

Comments This command is supported by devices that return true to the MCI_GETDEVCAPS_CAN_SAVE query.

See Also **MCI_LOAD**

MCI_SEEK

This MCI command message changes the current position of media as quickly as possible. Video and audio output are disabled during the seek. After the seek is complete, the device will be stopped. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_SEEK**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpSeek*.

MCI_WAIT

Specifies that the seek operation should finish before MCI returns control to the application.

MCI_SEEK_TO_END

Specifies to seek to the end of the media.

MCI_SEEK_TO_START

Specifies to seek to the start of the media.

MCI_TO

Specifies a position is included in the **dwTo** field of the **MCI_SEEK_PARMS** data structure. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command. Do not use this flag with MCI_SEEK_END or MCI_SEEK_START.

LPMCI_SEEK_PARMS *lpSeek*

Specifies a far pointer to the **MCI_SEEK_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MCI_SET_AUDIO

Specifies an audio channel number is included in the **dwAudio** field of the data structure identified by *lpSet*. This flag must be used with MCI_SET_ON or MCI_SET_OFF. Use one of the following constants to indicate the channel number:

MCI_SET_AUDIO_ALL

Specifies all audio channels.

MCI_SET_AUDIO_LEFT

Specifies the left channel.

MCI_SET_AUDIO_RIGHT

Specifies the right channel.

MCI_SET_DOOR_CLOSED

Instructs the device to close the media cover (if any).

MCI_SET_DOOR_OPEN

Instructs the device to open the media cover (if any).

MCI_SET_TIME_FORMAT

Specifies a time format parameter is included in the **dwTimeFormat** field of the data structure identified by *lpSet*. Specifying MCI_FORMAT_MILLISECONDS indicates that subsequent commands that specify time will use milliseconds for both input and output. Other units are device dependent.

MCI_SET_VIDEO

Sets the video signal on or off. This flag must be used with either MCI_SET_ON or MCI_SET_OFF. Devices that do not have video return MCIERR_UNSUPPORTED_FUNCTION.

MCI_SET_ON

Enables the specified video or audio channel.

MCI_SET_OFF

Disables the specified video or audio channel.

LPMCI_SET_PARMS *lpSet*

Specifies a far pointer to the **MCI_SET_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

MIDI Sequencer Extensions

The following additional command options are used with the "sequencer" device type:

DWORD *dwFlags*

The following additional flags apply to MIDI sequencer devices:

MCI_SEQ_SET_MASTER

Sets the sequencer as a source of synchronization data and indicates that the type of synchronization is specified in the **dwMaster** field of the data structure identified by *lpSet*.

MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION.

The following constants are defined for the synchronization type:

MCI_SEQ_MIDI

The sequencer will send MIDI format synchronization data.

MCI_SEQ_SMPTE

The sequencer will send SMPTE format synchronization data.

MCI_SEQ_NONE

The sequencer will not send synchronization data.

MCI_SEQ_SET_OFFSET

Changes the SMPTE offset of a sequence to that specified by the **dwOffset** field of the data structure identified by *lpSet*. This only affects sequences with a SMPTE division type.

MCI_SEQ_SET_PORT

Sets the output MIDI port of a sequence to that specified by the MIDI device ID in the **dwPort** field of the data structure identified by *lpSet*. The device will close the previous port (if any), and attempt to open and use the new port. If it fails, it will return an error and re-open the previously used port (if any). The following constants are defined for the ports:

MCI_SEQ_NONE

Closes the previously used port (if any). The sequencer will behave exactly the same as if a port were open, except no MIDI message will be sent.

MIDI_MAPPER

Sets the port opened to the MIDI Mapper.

MCI_FORMAT_SMPTE_30DROP
Sets the time format to 30 drop-frame SMPTE.

MCI_SEQ_FORMAT_SONGPTR
Sets the time format to song-pointer units.

LPMCI_SEQ_SET_PARMS *lpSet*
Specifies a far pointer to the **MCI_SEQ_SET_PARMS** data structure.

Videodisc Extensions

The following additional command options are used with the "videodisc" device type:

DWORD *dwFlags*
The following additional flags apply to videodisc devices:

MCI_SET_TIME_FORMAT
Specifies a time format parameter is included in the **dwTimeFormat** field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_CHAPTERS
Changes the time format to chapters.

MCI_FORMAT_FRAMES
Changes the time format to frames.

MCI_FORMAT_HMS
Changes the time format to hours, minutes, and seconds.

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds for both input and output.

MCI_VD_FORMAT_TRACK
Changes the time format to tracks. MCI uses continuous track numbers.

LPMCI_SET_PARMS *lpSet*
Specifies a far pointer to the **MCI_VD_SET_PARMS** structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

MCI_WAVE_SET_FORMATTAG

Sets the format type used for playing, recording, and saving to the **wFormatTag** field of the data structure identified by *lpSet*. Specifying WAVE_FORMAT_PCM changes the format to PCM.

MCI_WAVE_SET_SAMPLESPERSEC

Sets the samples per second used for playing, recording, and saving to the **nSamplesPerSec** field of the data structure identified by *lpSet*.

MCI_SET_TIME_FORMAT

Specifies a time format parameter is included in the **dwTimeFormat** field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_BYTES

Within a PCM data format, changes the time field description to bytes for input or output.

MCI_FORMAT_MILLISECONDS

Changes the time format to milliseconds for input or output.

MCI_FORMAT_SAMPLES

Changes the time format to samples for input or output.

LPMCI_WAVE_SET_PARMS *lpSet*

Specifies a far pointer to the **MCI_WAVE_SET_PARMS** data structure. This parameter replaces the standard default parameter data structure identified by *lpDefault*.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Parameters DWORD *dwFlags*

The following standard and command-specific flags apply to all devices:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpStatus*.

MCI_WAIT

Specifies that the status operation should finish before MCI returns control to the application.

MCI_STATUS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following constants define which status item to return in the **dwReturn** field of the data structure:

MCI_STATUS_CURRENT_TRACK

The **dwReturn** field is set to the current track number. MCI uses continuous track numbers.

MCI_STATUS_LENGTH

The **dwReturn** field is set to the total media length.

MCI_STATUS_MODE

The **dwReturn** field is set to the current mode of the device. The modes include the following:

- MCI_MODE_NOT_READY
- MCI_MODE_PAUSE
- MCI_MODE_PLAY
- MCI_MODE_STOP
- MCI_MODE_OPEN
- MCI_MODE_RECORD
- MCI_MODE_SEEK

MCI_STATUS_NUMBER_OF_TRACKS

The **dwReturn** field is set to the total number of playable tracks.

MCI_STATUS_POSITION

The **dwReturn** field is set to the current position.

Animation Extensions

The following additional command options are used with the "animation" device type:

DWORD *dwFlags*

The following extensions apply to the "animation" device type:

MCI_STATUS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for animation devices and indicate which item to return in the **dwReturn** field of the data structure:

MCI_ANIM_STATUS_FORWARD

The **dwReturn** field is set to TRUE if playing forward; otherwise, it is set to FALSE.

MCI_ANIM_STATUS_HPAL

The **dwReturn** field is set to the handle of the movie palette.

MCI_ANIM_STATUS_HWND

The **dwReturn** field is set to the handle of the playback window.

MCI_ANIM_STATUS_SPEED

The **dwReturn** field is set to the animation speed.

MCI_ANIM_STATUS_STRETCH

The **dwReturn** field is set to TRUE if stretching is enabled; otherwise, it is set to FALSE.

MCI_STATUS_MEDIA_PRESENT

The **dwReturn** field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*

Specifies a far pointer to the **MCI_STATUS_PARMS** data structure.

MCI_SEQ_STATUS_MASTER

The **dwReturn** field is set to the synchronization type used for master operation.

MCI_SEQ_STATUS_OFFSET

The **dwReturn** field is set to the current SMPTE offset of a sequence.

MCI_SEQ_STATUS_PORT

The **dwReturn** field is set to the MIDI device ID for the current port used by the sequence.

MCI_SEQ_STATUS_SLAVE

The **dwReturn** field is set to the synchronization type used for slave operation.

MCI_SEQ_STATUS_TEMPO

The **dwReturn** field is set to the current tempo of a MIDI sequence in beats-per-minute for PPQN files, or frames-per-second for SMPTE files.

MCI_STATUS_MEDIA_PRESENT

The **dwReturn** field is set to TRUE if the media for the device is present; otherwise, it is set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*

Specifies a far pointer to the **MCI_STATUS_PARMS** data structure. This parameter replaces the standard default parameter data structure.

MCI_VD_STATUS_SIDE

The **dwReturn** field is set to 1 or 2 to indicate which side of the disc is loaded. Not all videodisc devices support this flag.

MCI_VD_STATUS_SPEED

The **dwReturn** field is set to the play speed in frames per second.

MCIPIONR returns MCIERR_UNSUPPORTED_FUNCTION.

LPMCI_STATUS_PARMS *lpStatus*

Specifies a far pointer to the **MCI_STATUS_PARMS** data structure. This parameter replaces the standard default parameter data structure.

Waveform Audio Extensions

The following additional command options are used with the "waveaudio" device type:

DWORD *dwFlags*

The following additional flags apply to waveform audio devices:

MCI_STATUS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for waveform audio devices and indicate which item to return in the **dwReturn** field of the data structure:

MCI_STATUS_MEDIA_PRESENT

The **dwReturn** field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

MCI_WAVE_INPUT

The **dwReturn** field is set to the wave input device used for recording. If no device is in use and no device has been explicitly set, then the error return is MCI_WAVE_INPUTUNSPECIFIED.

MCI_WAVE_OUTPUT

The **dwReturn** field is set to the wave output device used for playing. If no device is in use and no device has been explicitly set, then the error return is MCI_WAVE_OUTPUTUNSPECIFIED.

MCI_WAVE_STATUS_AVGBYTESPERSEC

The **dwReturn** field is set to the current bytes per second used for playing, recording, and saving.

MCI_WAVE_STATUS_BITSPERSAMPLE

The **dwReturn** field is set to the current bits per sample used for playing, recording, and saving PCM formatted data.

Video Overlay Extensions

The following additional command options are used with the "overlay" device type:

DWORD *dwFlags*

The following additional flags apply to video overlay devices:

MCI_OVLY_STATUS_HWND

The **dwReturn** field is set to the handle of the window associated with the video overlay device.

MCI_OVLY_STATUS_STRETCH

The **dwReturn** field is set to TRUE if stretching is enabled; otherwise, it is set to FALSE.

MCI_STATUS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for video overlay devices and indicate which item to return in the **dwReturn** field of the data structure:

MCI_STATUS_MEDIA_PRESENT

The **dwReturn** field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*

Specifies a far pointer to the **MCI_STATUS_PARMS** data structure.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_STEP

This MCI command message steps the player one or more frames.

Parameters DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_STEP**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpStep*.

MCI_WAIT

Specifies that the step operation should finish before MCI returns control to the application.

MCI_STOP

This MCI command message stops all play and record sequences, unloads all play buffers, and ceases display of video images. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_STOP**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpStop*.

MCI_WAIT

Specifies that the device should stop before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpStop*

Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

The difference between **MCI_STOP** and **MCI_PAUSE** depends upon the device. If possible, **MCI_PAUSE** suspends device operation but leaves the device ready to resume play immediately.

See Also

MCI_PAUSE, MCI_PLAY, MCI_RECORD, MCI_RESUME

MCI_UNFREEZE (VIDEO OVERLAY)

This MCI command message restores motion to an area of the video buffer frozen with **MCI_FREEZE**. This command is part of the video overlay command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_UNFREEZE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpFreeze*.

MCI_WAIT

Specifies that the unfreeze operation should finish before MCI returns control to the application.

MCI_OVLY_RECT

Specifies that the **rc** field of the data structure identified by *lpFreeze* contains a valid display rectangle. This is a required parameter.

LPMCI_OVLY_RECT_PARMS *lpFreeze*

Specifies a far pointer to a **MCI_OVLY_RECT_PARMS** data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command applies to video overlay devices.

See Also

MCI_FREEZE

MCI_WHERE (ANIMATION/VIDEO OVERLAY)

This MCI command message obtains the clipping rectangle for the video device. The top and left fields of the returned rectangle contain the origin of the clipping rectangle, and the right and bottom fields contain the width and height of the clipping rectangle. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_WHERE**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpQuery*.

MCI_WAIT

Specifies that the operation should complete before MCI returns control to the application.

DWORD *lpQuery*

Specifies a far pointer to a device-specific data structure. For a description of this parameter, see the *lpQuery* description included with the device extensions.

Animation Extensions

The following additional command options are used with the "animation" device type:

DWORD *dwFlags*

The following additional flags apply to animation devices supporting **MCI_WHERE**:

MCI_ANIM_WHERE_DESTINATION

Obtains the destination display rectangle. The rectangle coordinates are placed in the **rc** field of the data structure identified by *lpQuery*.

MCI_ANIM_WHERE_SOURCE

Obtains the animation source rectangle. The rectangle coordinates are placed in the **rc** field of the data structure identified by *lpQuery*.

LPMCI_ANIM_RECT_PARMS *lpQuery*

Specifies a far pointer to a **MCI_ANIM_RECT_PARMS** data structure.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_WINDOW**:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lpWindow*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

DWORD *lpWindow*

Specifies a far pointer to a device specific data structure. For a description of this parameter, see the *lpWindow* description included with the device extensions.

Animation Extensions

The following additional command options are used with the "animation" device type:

DWORD *dwFlags*

The following additional flags apply to animation devices supporting **MCI_WINDOW**:

MCI_ANIM_WINDOW_DISABLE_STRETCH

Disables stretching of the image.

MCI_ANIM_WINDOW_ENABLE_STRETCH

Enables stretching of the image.

MCI_ANIM_WINDOW_HWND

Indicates the handle of the window to use for the destination is included in the **hWnd** field of the data structure identified by *lpWindow*. Set this to MCI_ANIM_WINDOW_DEFAULT to return to the default window.

MCI_ANIM_WINDOW_STATE

Indicates the **nCmdShow** field of the **MCI_ANIM_WINDOW_PARMS** data structure contains parameters for setting the window state. This flag is equivalent to calling **ShowWindow** with the state parameter. The constants are the same those in WINDOWS.H (such as SW_HIDE, SW_MINIMIZE, or SW_SHOWNORMAL).

MCI_ANIM_WINDOW_TEXT

Indicates the **lpstrText** field of the **MCI_ANIM_WINDOW_PARMS** data structure contains a pointer to a buffer containing the caption used for the window.

MIM_CLOSE

This message is sent to a MIDI input callback function when a MIDI input device is closed. The device handle is no longer valid once this message has been sent.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MIM_CLOSE

MIM_DATA

This message is sent to a MIDI input callback function when a MIDI message is received by a MIDI input device.

Parameters

DWORD *dwParam1*

Specifies the MIDI message that was received. The message is packed into a DWORD with the first byte of the message in the low-order byte.

DWORD *dwParam2*

Specifies the time that the message was received by the input device driver. The time stamp is specified in milliseconds, beginning at 0 when **midiInStart** was called.

Return Value

None.

Comments

MIDI messages received from a MIDI input port have running status disabled; each message is expanded to include the MIDI status byte.

This message is not sent when a MIDI system-exclusive message is received.

See Also

MM_MIM_DATA, MIM_LONGDATA

MIM_LONGERROR

This message is sent to a MIDI input callback function when an invalid MIDI system-exclusive message is received.

Parameters

DWORD *dwParam1*

Specifies a pointer to a **MIDIHDR** structure identifying the buffer containing the invalid message.

DWORD *dwParam2*

Specifies the time that the data was received by the input device driver. The time stamp is specified in milliseconds, beginning at 0 when **midiInStart** was called.

Return Value

None.

Comments

The returned buffer might not be full. The **dwBytesRecorded** field of the **MIDIHDR** structure specified by *dwParam1* will specify the number of bytes recorded into the buffer.

See Also

MM_MIM_LONGERROR

MIM_OPEN

This message is sent to a MIDI input callback function when a MIDI input device is opened.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MIM_OPEN

JOY_BUTTON3CHG
Set if third joystick button has changed.

JOY_BUTTON4CHG
Set if fourth joystick button has changed.

LPARAM *lParam*
The low-order word contains the current *x*-position of the joystick. The high-order word contains the current *y*-position.

Return Value None.

See Also **MM_JOY1BUTTONDOWN**

MM_JOY1MOVE

This message is sent to the window that has captured joystick 1 when the joystick position changes.

Parameters WPARAM *wParam*
Indicates which joystick buttons are pressed. It can be any combination of the following values:

JOY_BUTTON1
Set if first joystick button is pressed.

JOY_BUTTON2
Set if second joystick button is pressed.

JOY_BUTTON3
Set if third joystick button is pressed.

JOY_BUTTON4
Set if fourth joystick button is pressed.

LPARAM *lParam*
The low-order word contains the current *x*-position of the joystick. The high-order word contains the current *y*-position.

Return Value None.

See Also **MM_JOY1ZMOVE**

JOY_BUTTON3CHG
Set if third joystick button has changed.

JOY_BUTTON4CHG
Set if fourth joystick button has changed.

LPARAM *lParam*
The low-order word contains the current *x*-position of the joystick. The high-order word contains the current *y*-position.

Return Value None.

See Also **MM_JOY2BUTTONUP**

MM_JOY2BUTTONUP

This message is sent to the window that has captured joystick 2 when a button is released.

Parameters WPARAM *wParam*
Indicates which button has changed state. It can be any one of the following combined with any of the flags defined in MM_JOY1MOVE.

JOY_BUTTON1CHG
Set if first joystick button has changed.

JOY_BUTTON2CHG
Set if second joystick button has changed.

JOY_BUTTON3CHG
Set if third joystick button has changed.

JOY_BUTTON4CHG
Set if fourth joystick button has changed.

LPARAM *lParam*
The low-order word contains the current *x*-position of the joystick. The high-order word contains the current *y*-position.

Return Value None.

See Also **MM_JOY2BUTTONDOWN**

JOY_BUTTON3
Set if third joystick button is pressed.

JOY_BUTTON4
Set if fourth joystick button is pressed.

LPARAM *lParam*
The low-order word contains the current *z*-position of the joystick.

Return Value None.

See Also **MM_JOY2MOVE**

MM_MCINOTIFY

This message is sent to a window to notify an application that an MCI device has completed an operation. MCI devices send this message only when the MCI_NOTIFY flag is used with an MCI command message or when the **notify** flag is used with an MCI command string.

Parameters WPARAM *wParam*
Contains one of the following flags:

MCI_NOTIFY_ABORTED
Specifies that the device received a command that prevented the current conditions for initiating the callback from being met. If a new command interrupts the current command and it also requests notification, the device sends only this message and not MCI_NOTIFY_SUPERCEDED.

MCI_NOTIFY_SUCCESSFUL
Specifies that the conditions initiating the callback have been met.

MCI_NOTIFY_SUPERSEDED
Specifies that the device received another command with the MCI_NOTIFY flag set and the current conditions for initiating the callback have been superseded.

MCI_NOTIFY_FAILURE
Specifies that a device error occurred while the device was executing the command.

LPARAM *lParam*
The low-order word specifies the ID of the device initiating the callback.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MM_MIM_DATA

This message is sent to a window when a MIDI message is received by a MIDI input device.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that received the MIDI message.

LPARAM *lParam*

Specifies the MIDI message that was received. The message is packed into a DWORD with the first byte of the message in the low-order byte.

Return Value

None.

Comments

MIDI messages received from a MIDI input port have running status disabled; each message is expanded to include the MIDI status byte.

This message is not sent when a MIDI system-exclusive message is received. No time stamp is available with this message. For time-stamped input data, you must use the messages that are sent to low-level callback functions.

See Also

MIM_DATA, MM_MIM_LONGDATA

MM_MIM_ERROR

This message is sent to a window when an invalid MIDI message is received.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that received the invalid message.

LPARAM *lParam*

Specifies the invalid MIDI message. The message is packed into a DWORD with the first byte of the message in the low-order byte.

Return Value

None.

See Also

MIM_ERROR

MM_MIM_OPEN

This message is sent to a window when a MIDI input device is opened.

Parameters

WPARAM *wParam*
Specifies the handle to the MIDI input device that was opened.

LPARAM *lParam*
Not used.

Return Value None.

See Also **MIM_OPEN**

MM_MOM_CLOSE

This message is sent to a window when a MIDI output device is closed. The device handle is no longer valid once this message has been sent.

Parameters

WPARAM *wParam*
Specifies the handle to the MIDI output device.

LPARAM *lParam*
Not used.

Return Value None.

See Also **MOM_CLOSE**

MM_MOM_DONE

This message is sent to a window when the specified system-exclusive buffer has been played and is being returned to the application.

Parameters

WPARAM *wParam*
Specifies a handle to the MIDI output device that played the buffer.

LPARAM *lParam*
Specifies a far pointer to a **MIDIHDR** structure identifying the buffer.

Return Value None.

See Also **MOM_DONE**

MM_WIM_OPEN

This message is sent to a window when a waveform input device is opened.

Parameters WPARAM *wParam*
Specifies a handle to the waveform input device that was opened.

LPARAM *lParam*
Not used.

Return Value None.

See Also **WIM_OPEN**

MM_WOM_CLOSE

This message is sent to a window when a waveform output device is closed. The device handle is no longer valid once this message has been sent.

Parameters WPARAM *wParam*
Specifies a handle to the waveform output device that was closed.

LPARAM *lParam*
Not used.

Return Value None.

See Also **WOM_CLOSE**

MM_WOM_DONE

This message is sent to a window when the specified output buffer is being returned to the application. Buffers are returned to the application when they have been played, or as the result of a call to **waveOutReset**.

Parameters WPARAM *wParam*
Specifies a handle to the waveform output device that played the buffer.

LPARAM *lParam*
Specifies a far pointer to a **WAVEHDR** structure identifying the buffer.

Return Value None.

See Also **WOM_DONE**

Return Value The return value is zero if the operation is successful. Otherwise, the return value specifies an error value. Possible error returns are:

MMIOM_CANNOTOPEN
Specified file could not be opened.

MMIOM_OUTOFMEMORY
Not enough memory to perform operation.

Comments The **dwFlags** field of the **MMIOINFO** structure contains option flags passed to the **mmioOpen** function. The **lDiskOffset** field of the **MMIOINFO** structure is initialized to zero. If this value is incorrect, then the I/O procedure must correct it.

If the caller passed a **MMIOINFO** structure to **mmioOpen**, the return value will be returned in the **wErrorRet** field.

See Also **mmioOpen**, **MMIOM_CLOSE**

MMIOM_READ

This message is sent to an I/O procedure by **mmioRead** to request that a specified number of bytes be read from an open file.

Parameters LPARAM *lParam1*
Specifies a huge pointer to the buffer to be filled with data read from the file.

LPARAM *lParam2*
Specifies the number of bytes to read from the file.

Return Value The return value is the number of bytes actually read from the file. If no more bytes can be read, the return value is zero. If there is an error, the return value is –1.

Comments The I/O procedure is responsible for updating the **lDiskOffset** field of the **MMIOINFO** structure to reflect the new file position after the read operation.

See Also **mmioRead**, **MMIOM_WRITE**, **MMIOM_WRITEFLUSH**

MMIOM_WRITE

This message is sent to an I/O procedure by **mmioWrite** to request that data be written to an open file.

Parameters

LPARAM *lParam1*

Specifies a huge pointer to a buffer containing the data to write to the file.

LPARAM *lParam2*

Specifies the number of bytes to write to the file.

Return Value

The return value is the number of bytes actually written to the file. If there is an error, the return value is –1.

Comments

The I/O procedure is responsible for updating the **lDiskOffset** field of the **MMIOINFO** structure to reflect the new file position after the write operation.

See Also

mmioWrite, **MMIOM_READ**, **MMIOM_WRITEFLUSH**

MMIOM_WRITEFLUSH

This message is sent to an I/O procedure by **mmioWrite** to request that data be written to an open file and then that any internal buffers used by the I/O procedure be flushed to disk.

Parameters

LPARAM *lParam1*

Specifies a huge pointer to a buffer containing the data to write to the file.

LPARAM *lParam2*

Specifies the number of bytes to write to the file.

Return Value

The return value is the number of bytes actually written to the file. If there is an error, the return value is –1.

Comments

The I/O procedure is responsible for updating the **lDiskOffset** field of the **MMIOINFO** structure to reflect the new file position after the write operation.

Note that this message is equivalent to the **MMIOM_WRITE** message except that it additionally requests that the I/O procedure flush its internal buffers, if any. Unless an I/O procedure performs internal buffering, this message can be handled exactly like the **MMIOM_WRITE** message.

See Also

mmioWrite, **mmioFlush**, **MMIOM_READ**, **MMIOM_WRITE**

WIM_CLOSE

This message is sent to a waveform input callback function when a waveform input device is closed. The device handle is no longer valid once this message has been sent.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

None.

See Also

MM_WIM_CLOSE

WIM_DATA

This message is sent to a waveform input callback function when waveform data is present in the input buffer and the buffer is being returned to the application. The message can be sent either when the buffer is full, or after the **waveInReset** function is called.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **WAVEHDR** structure identifying the buffer containing the waveform data.

DWORD *dwParam2*
Not used.

Return Value

None.

Comments

The returned buffer might not be full. Use the **dwBytesRecorded** field of the **WAVEHDR** structure specified by *dwParam1* to determine the number of bytes recorded into the returned buffer.

See Also

MM_WIM_DATA

WOM_OPEN

This message is sent to a waveform output callback function when a waveform output device is opened.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

None.

See Also

MM_WOM_OPEN

Chapter 6
Data Types and Structures

This chapter describes the multimedia data types and data structures for Windows. For information about standard Windows data types, see the *Microsoft Windows Programmer's Reference, Volume 3: Messages, Structures, Macros*. This chapter contains three parts:

- An alphabetical list of all multimedia data types.
- An overview of all data structures, organized by category. This overview includes brief descriptions of all data structures.
- Detailed descriptions of all data structures, organized alphabetically. These descriptions list the structure definition and the type and contents of each field in the structure.

You can also refer to the MMSYSTEM.H header file to see the actual data structure definitions.

Data Structure Overview

The multimedia data structures are grouped as follows:

- Auxiliary audio data structure
- Joystick data structures
- Media Control Interface (MCI) data structures
- MIDI audio data structures
- Multimedia file I/O data structures
- Timer data structures
- Waveform audio data structures

Each data structure has an associated long pointer data type with prefix LP.

Auxiliary Audio Data Structure

The following data structure is used with auxiliary audio devices:

AUXCAPS
A data structure that describes the capabilities of an auxiliary audio device.

Joystick Data Structures

The following data structures are used with joystick functions:

JOYCAPS
A data structure that defines joystick capabilities.

JOYINFO
A data structure for joystick information.

MCI_OPEN_PARMS
MCI_ANIM_OPEN_PARMS (animation device)
MCI_OVLY_OPEN_PARMS (video overlay device)
MCI_WAVE_OPEN_PARMS (waveform audio device)
Data structures that specify parameters for the **MCI_OPEN** command.

MCI_STATUS_PARMS
A data structure that specifies parameters for the **MCI_STATUS** command.

Data Structures for MCI Basic Commands

The following data structures are used to specify parameter blocks for basic command messages (messages that, if recognized by an MCI device, have a standard set of basic options):

MCI_GENERIC_PARMS
A data structure that specifies parameters for the **MCI_PAUSE**, **MCI_RESUME**, and **MCI_STOP** commands.

MCI_LOAD_PARMS
MCI_OVLY_LOAD_PARMS (video overlay device)
A data structure that specifies parameters for the **MCI_LOAD** command.

MCI_PLAY_PARMS
MCI_ANIM_PLAY_PARMS (animation device)
MCI_VD_PLAY_PARMS (videodisc device)
Data structures that specify parameters for the **MCI_PLAY** command.

MCI_RECORD_PARMS
A data structure that specifies parameters for the **MCI_RECORD** command.

MCI_SAVE_PARMS
MCI_OVLY_SAVE_PARMS (video overlay device)
A data structure that specifies parameters for the **MCI_SAVE** command.

MCI_SEEK_PARMS
A data structure that specifies parameters for the **MCI_SEEK** command.

MCI_SET_PARMS
MCI_SEQ_SET_PARMS (sequencer device)
MCI_WAVE_SET_PARMS (waveform audio device)
Data structures that specify parameters for the **MCI_SET** command.

MIDI Audio Data Structures

The following data structures are used with MIDI functions:

MIDIHDR
A data structure representing a header for MIDI input and output data blocks.

MIDIINCAPS
A data structure that describes the capabilities of a MIDI input device.

MIDIOUTCAPS
A data structure that describes the capabilities of a MIDI output device.

Multimedia File I/O Data Structures

The following data structures are used with the multimedia file I/O functions:

MMIOINFO
A data structure for information about an open file.

MMCKINFO
A data structure for information about a RIFF chunk in an open file.

Timer Data Structures

The following data structures are used with timer functions:

MMTIME
A data structure that represents time in one of several different formats.

TIMECAPS
A data structure that defines timer capabilities.

Data Structures Reference

This section lists the multimedia data structures alphabetically. Each structure description shows the definition of the structure type and a description of each structure field.

AUXCAPS

The **AUXCAPS** structure describes the capabilities of an auxiliary output device.

```
typedef struct auxcaps_tag {
    UINT  wMid;
    UINT  wPid;
    VERSION  vDriverVersion;
    char  szPname[MAXPNAMELEN];
    UINT  wTechnology;
    DWORD  dwSupport;
} AUXCAPS;
```

Fields

The **AUXCAPS** structure has the following fields:

wMid
Specifies a manufacturer ID for the device driver for the auxiliary audio device. Manufacturer IDs are listed in Appendix B, "Manufacturer ID and Product ID Lists."

wPid
Specifies a product ID for the auxiliary audio device. Product IDs are listed in Appendix B, "Manufacturer ID and Product ID Lists."

vDriverVersion
Specifies the version number of the device driver for the auxiliary audio device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]
Specifies the product name in a NULL-terminated string.

wTechnology
Describes the type of the auxiliary audio output according to one of the following flags:

AUXCAPS_CDAUDIO
Audio output from an internal CD-ROM drive.

AUXCAPS_AUXIN
Audio output from auxiliary input jacks.

wXmin
Specifies the minimum *x*-position value of the joystick.

wXmax
Specifies the maximum *x*-position value of the joystick.

wYmin
Specifies the minimum *y*-position value of the joystick.

wYmax
Specifies the maximum *y*-position value of the joystick.

wZmin
Specifies the minimum *z*-position value of the joystick.

wZmax
Specifies the maximum *z*-position value of the joystick.

wNumButtons
Specifies the number of buttons on the joystick.

wPeriodMin
Specifies the smallest polling interval supported when captured by **joySetCapture**.

wPeriodMax
Specifies the largest polling interval supported when captured by **joySetCapture**.

See Also **joyGetDevCaps**

JOYINFO

Structure for storing joystick position and button state information.

```
typedef struct joyinfo_tag {
    UINT  wXpos;
    UINT  wYpos;
    UINT  wZpos;
    UINT  wButtons;
} JOYINFO;
```

Fields The **JOYINFO** structure has the following fields:

wXpos
Specifies the current *x*-position of joystick.

wYpos
Specifies the current *y*-position of joystick.

Fields

The **MCI_ANIM_OPEN_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

wDeviceID
Specifies the device ID returned to user.

wReserved0
Reserved field.

lpstrDeviceType
Specifies the name or constant ID of the device type.

lpstrElementName
Specifies the device element name (usually a path).

lpstrAlias
Specifies an optional device alias.

dwStyle
Specifies the window style.

hWndParent
Specifies the handle to use as the window parent.

wReserved1
Reserved.

See Also

MCI_OPEN

MCI_ANIM_PLAY_PARMS

The **MCI_ANIM_PLAY_PARMS** structure contains parameters for the **MCI_PLAY** message for animation devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields. You can use the **MCI_PLAY_PARMS** data structure in place of **MCI_ANIM_PLAY_PARMS** if you are not using the extended data fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
    DWORD  dwSpeed;
} MCI_ANIM_PLAY_PARMS;
```

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrames;
} MCI_ANIM_STEP_PARMS;
```

Fields The **MCI_ANIM_STEP_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrames
Specifies the number of frames to step.

See Also **MCI_STEP**

MCI_ANIM_UPDATE_PARMS

The **MCI_ANIM_UPDATE_PARMS** structure contains parameters for the **MCI_UPDATE** message for animation devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    RECT  rc;
    HDC  hDC;
} MCI_ANIM_UPDATE_PARMS;
```

Fields The **MCI_ANIM_UPDATE_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Specifies a window rectangle.

hDC
Specifies a handle to the device context.

See Also **MCI_UPDATE**

MCI_BREAK_PARMS

The **MCI_BREAK_PARMS** structure contains parameters for the **MCI_BREAK** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    int  nVirtKey;
    UINT  wReserved0;
    HWND  hwndBreak;
    UINT  wReserved1;
} MCI_BREAK_PARMS;
```

Fields

The **MCI_BREAK_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

nVirtKey
Specifies the virtual key code used for the break key.

wReserved0
Reserved.

hwndBreak
Specifies a window handle of the window that must be the current window for break detection.

wReserved1
Reserved.

See Also

MCI_BREAK

MCI_INFO_PARMS

The **MCI_INFO_PARMS** structure contains parameters for the **MCI_INFO** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    LPSTR  lpstrReturn;
    DWORD  dwRetSize;
} MCI_INFO_PARMS;
```

Fields

The **MCI_INFO_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrReturn
Specifies a long pointer to a user-supplied buffer for the return string.

dwRetSize
Specifies the size in bytes of the buffer for the return string.

See Also

MCI_INFO

MCI_LOAD_PARMS

The **MCI_LOAD_PARMS** structure contains the information for **MCI_LOAD** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    LPCSTR  lpfilename;
} MCI_LOAD_PARMS;
```

Fields

The **MCI_LOAD_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpfilename
Specifies a far pointer to a null-terminated string containing the filename of the device element to load.

See Also

MCI_LOAD

MCI_OVLY_LOAD_PARMS

The **MCI_OVLY_LOAD_PARMS** structure contains parameters for the MCI_LOAD message for video overlay devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    LPCSTR  lpfilename;
    RECT  rc;
} MCI_OVLY_LOAD_PARMS;
```

Fields

The **MCI_OVLY_LOAD_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpfilename
Specifies a far pointer to the buffer containing a null-terminated string.

rc
Specifies a rectangle.

See Also

MCI_LOAD

MCI_OVLY_OPEN_PARMS

The **MCI_OVLY_OPEN_PARMS** structure contains information for **MCI_OPEN** message for video overlay devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields. You can use the **MCI_OPEN_PARMS** data structure in place of **MCI_OVLY_OPEN_PARMS** if you are not using the extended data fields.

```
typedef struct {
    DWORD  dwCallback;
    UINT  wDeviceID;
    UINT  wReserved0;
    LPCSTR  lpstrDeviceType;
    LPCSTR  lpstrElementName;
    LPCSTR  lpstrAlias;
    DWORD  dwStyle;
    DWORD  hWndParent;
    UINT  wReserved1;
} MCI_OVLY_OPEN_PARMS;
```

Fields The **MCI_OVLY_RECT_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Specifies a rectangle.

See Also **MCI_PUT, MCI_WHERE**

MCI_OVLY_SAVE_PARMS

The **MCI_OVLY_SAVE_PARMS** structure contains parameters for the **MCI_SAVE** message for video overlay devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    LPCSTR  lpfilename;
    RECT  rc;
} MCI_OVLY_SAVE_PARMS;
```

Fields The **MCI_OVLY_SAVE_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpfilename
Specifies a far pointer to the buffer containing a null-terminated string.

rc
Specifies a rectangle.

See Also **MCI_SAVE**

MCI_PLAY_PARMS

The **MCI_PLAY_PARMS** structure contains parameters for the **MCI_PLAY** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
} MCI_PLAY_PARMS;
```

Fields

The **MCI_PLAY_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom
Specifies the position to play from.

dwTo
Specifies the position to play to.

See Also

MCI_PLAY

MCI_RECORD_PARMS

The **MCI_RECORD_PARMS** structure contains parameters for the **MCI_RECORD** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
} MCI_RECORD_PARMS;
```

MCI_SEEK_PARMS

The **MCI_SEEK_PARMS** structure contains parameters for the **MCI_SEEK** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwTo;
} MCI_SEEK_PARMS;
```

Fields

The **MCI_SEEK_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTo
Specifies the position to seek to.

See Also

MCI_SEEK

MCI_SEQ_SET_PARMS

The **MCI_SEQ_SET_PARMS** structure contains parameters for the **MCI_SET** message for MIDI sequencer devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwTimeFormat;
    DWORD  dwAudio;
    DWORD  dwTempo;
    DWORD  dwPort;
    DWORD  dwSlave;
    DWORD  dwMaster;
    DWORD  dwOffset;
} MCI_SEQ_SET_PARMS;
```

Fields The **MCI_SET_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeFormat
Specifies the time format used by the device.

dwAudio
Specifies the audio output channel.

See Also MCI_SET

MCI_STATUS_PARMS

The **MCI_STATUS_PARMS** structure contains parameters for the **MCI_STATUS** message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwReturn;
    DWORD  dwItem;
    DWORD  dwTrack;
} MCI_STATUS_PARMS;
```

Fields The **MCI_STATUS_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwReturn
Contains the return information on exit.

dwItem
Identifies the capability being queried.

dwTrack
Specifies the length or number of tracks.

See Also MCI_STATUS

MCI_VD_ESCAPE_PARMS

The **MCI_VD_ESCAPE_PARMS** structure contains parameters for the **MCI_ESCAPE** message for videodisc devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    LPCSTR  lpstrCommand;
} MCI_VD_ESCAPE_PARMS;
```

Fields

The **MCI_VD_ESCAPE_PARMS** structure has the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrCommand

Specifies a far pointer to a null-terminated buffer containing the command to send to the device.

See Also

MCI_ESCAPE

MCI_VD_PLAY_PARMS

The **MCI_VD_PLAY_PARMS** structure contains parameters for the **MCI_PLAY** message for videodiscs. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields. You can use the **MCI_PLAY_PARMS** data structure in place of **MCI_VD_PLAY_PARMS** if you are not using the extended data fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
    DWORD  dwSpeed;
} MCI_VD_PLAY_PARMS;
```

MCI_WAVE_DELETE_PARMS

The **MCI_WAVE_DELETE_PARMS** structure contains parameters for the **MCI_DELETE** message for waveform audio devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
} MCI_WAVE_DELETE_PARMS;
```

Fields

The **MCI_WAVE_DELETE_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom
Specifies the starting position for the delete.

dwTo
Specifies the end position for the delete.

See Also

MCI_DELETE

MCI_WAVE_OPEN_PARMS

The **MCI_WAVE_OPEN_PARMS** structure contains information for **MCI_OPEN** message for waveform audio devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of **mciSendCommand** to validate the fields. You can use the **MCI_OPEN_PARMS** data structure in place of **MCI_WAVE_OPEN_PARMS** if you are not using the extended data fields.

```
typedef struct {
    DWORD  dwCallback;
    UINT  wDeviceID;
    UINT  wReserved0;
    LPCSTR  lpstrDeviceType;
    LPCSTR  lpstrElementName;
    LPCSTR  lpstrAlias;
    DWORD  dwBufferSeconds;
} MCI_WAVE_OPEN_PARMS;
```

```
    UINT  wReserved4;
    UINT  wBitsPerSample;
    UINT  wReserved5;
} MCI_WAVE_SET_PARMS;
```

Fields

The **MCI_WAVE_SET_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeFormat
Specifies the time format used by the device.

dwAudio
Specifies the channel used for audio output.

wInput
Specifies the channel used for audio input.

wReserved0
Reserved.

wOutput
Specifies the channel used for output.

wReserved1
Reserved.

wFormatTag
Specifies the interpretation of the waveform data.

wReserved2
Reserved.

nChannels
Specifies mono (1) or stereo (2).

wReserved3
Reserved.

nSamplesPerSec
Specifies the samples per second used for the waveform.

nAvgBytesPerSec
Specifies the sample rate in bytes per second.

dwFlags

Specifies flags giving information about the data buffer.

MHDR_DONE

Set by the device driver to indicate that it is finished with the data buffer and is returning it to the application.

MHDR_PREPARED

Set by Windows to indicate that the data buffer has been prepared with **midiInPrepareHeader** or **midiOutPrepareHeader**.

lpNext

Reserved and should not be used.

reserved

Reserved and should not be used.

MIDIINCAPS

The **MIDIINCAPS** structure describes the capabilities of a MIDI input device.

```
typedef struct midiincaps_tag {
    UINT  wMid;
    UINT  wPid;
    VERSION  vDriverVersion;
    char  szPname[MAXPNAMELEN];
} MIDIINCAPS;
```

Fields

The **MIDIINCAPS** structure has the following fields:

wMid

Specifies a manufacturer ID for the device driver for the MIDI input device. Manufacturer IDs are defined in Appendix B, "Manufacturer ID and Product ID Lists."

wPid

Specifies a product ID for the MIDI input device. Product IDs are defined in Appendix B, "Manufacturer ID and Product ID Lists."

wTechnology

Describes the type of the MIDI output device according to one of the following flags:

MOD_MIDIPORT

Indicates the device is a MIDI hardware port.

MOD_SQSYNTH

Indicates the device is a square wave synthesizer.

MOD_FMSYNTH

Indicates the device is an FM synthesizer.

MOD_MAPPER

Indicates the device is the Microsoft MIDI Mapper.

wVoices

Specifies the number of voices supported by an internal synthesizer device. If the device is a port, the field is not meaningful and will be set to 0.

wNotes

Specifies the maximum number of simultaneous notes that may be played by an internal synthesizer device. If the device is a port, the field is not meaningful and will be set to 0.

wChannelMask

Specifies the channels that an internal synthesizer device responds to, where the least significant bit refers to channel 0 and the most significant bit to channel 15. Port devices transmit on all channels and so will set this field to 0xFFFF.

dwSupport

Specifies optional functionality supported by the device.

MIDICAPS_VOLUME

Supports volume control.

MIDICAPS_LRVOLUME

Supports separate left and right volume control.

MIDICAPS_CACHE

Supports patch caching.

MMIOINFO

This structure contains the current state of a file opened with **mmioOpen**.

```
typedef struct _MMIOINFO {
    DWORD  dwFlags;
    FOURCC  fccIOProc;
    LPMMIOPROC  pIOProc;
    UINT  wErrorRet;
    HTASK  htask;    LONG  cchBuffer;
    HPSTR  pchBuffer;
    HPSTR  pchNext;
    HPSTR  pchEndRead;
    HPSTR  pchEndWrite;
    LONG  lBufOffset;
    LONG  lDiskOffset;
    DWORD  adwInfo[4];
    DWORD  dwReserved1;
    DWORD  dwReserved2;
    HMMIO  hmmio;
} MMIOINFO;
```

Fields

The **MMIOINFO** structure has the following fields:

dwFlags

Specifies options indicating how a file was opened:

MMIO_READ
: The file was opened only for reading.

MMIO_WRITE
: The file was opened only for writing.

MMIO_READWRITE
: The file was opened for both reading and writing.

MMIO_COMPAT
: The file was opened with compatibility mode, allowing any process on a given computer to open the file any number of times.

MMIO_EXCLUSIVE
: The file was opened with exclusive mode, denying other processes both read and write access to the file.

MMIO_DENYWRITE
: Other processes are denied write access to the file.

pchNext

Specifies a huge pointer to the next location in the I/O buffer to be read or written. If no more bytes can be read without calling **mmioAdvance** or **mmioRead**, then this field points to **pchEndRead**. If no more bytes can be written without calling **mmioAdvance** or **mmioWrite**, then this field points to **pchEndWrite**.

pchEndRead

Specifies a pointer to the location that is one byte past the last location in the buffer that can be read.

pchEndWrite

Specifies a pointer to the location that is one byte past the last location in the buffer that can be written.

lBufOffset

Reserved for internal use by MMIO functions.

lDiskOffset

Specifies the current file position. The current file position is an offset in bytes from the beginning of the file. I/O procedures are responsible for maintaining this field.

adwInfo[4]

Contained state information maintained by the I/O procedure. I/O procedures can also use these fields to transfer information from the caller to the I/O procedure when the caller opens a file.

dwReserved1

Reserved for internal use by MMIO functions.

dwReserved2

Reserved for internal use by MMIO functions.

hmmio

Specifies the MMIO handle to the open file. I/O procedures can use this handle when calling other MMIO functions.

See Also **mmioGetInfo**

u

The contents of the union. The following fields are contained in union **u**:

ms

Milliseconds. Used when **wType** is TIME_MS.

sample

Samples. Used when **wType** is TIME_SAMPLES.

cb

Byte count. Used when **wType** is TIME_BYTES.

smpte

SMPTE time. Used when **wType** is TIME_SMPTE. The following fields are contained in structure **smpte**:

hour

Hours.

min

Minutes.

sec

Seconds.

frame

Frames.

fps

Frames per second (24, 25, 29 [30 drop] or 30).

dummy

Dummy byte for alignment.

midi

MIDI time. Used when **wType** is TIME_MIDI. The following fields are contained in structure **midi**:

songptrpos

Song pointer position.

WAVEFORMAT

The **WAVEFORMAT** structure describes the format of waveform data. Only format information common to all waveform data formats is included in this structure. For formats that require additional information, this structure is included as a field in another data structure along with the additional information.

```
typedef struct waveformat_tag {
    WORD  wFormatTag;
    WORD  nChannels;
    DWORD  nSamplesPerSec;
    DWORD  nAvgBytesPerSec;
    WORD  nBlockAlign;
} WAVEFORMAT;
```

Fields

The **WAVEFORMAT** structure has the following fields:

wFormatTag

Specifies the format type. Currently defined format types are as follows:

WAVE_FORMAT_PCM
: Waveform data is PCM.

nChannels

Specifies the number of channels in the waveform data. Mono data uses 1 channel and stereo data uses 2 channels.

nSamplesPerSec

Specifies the sample rate in samples per second.

nAvgBytesPerSec

Specifies the required average data transfer rate in bytes per second.

nBlockAlign

Specifies the block alignment in bytes. The block alignment is the minimum atomic unit of data.

Comments

For PCM data, the block alignment is the number of bytes used by a single sample, including data for both channels if the data is stereo. For example, the block alignment for 16-bit stereo PCM is 4 bytes (2 channels, 2 bytes per sample).

See Also

PCMWAVEFORMAT

dwLoops

Specifies the number of times to play the loop. This parameter is used only with output data buffers.

lpNext

Reserved and should not be used.

reserved

Reserved and should not be used.

Comments

Use the WHDR_BEGINLOOP and WHDR_ENDLOOP flags in the **dwFlags** field to specify the beginning and ending data blocks for looping. To loop on a single block, specify both flags for the same block. Use the **dwLoops** field in the **WAVEHDR** structure for the first block in the loop to specify the number of times to play the loop.

WAVEINCAPS

The **WAVEINCAPS** structure describes the capabilities of a waveform input device.

```
typedef struct waveincaps_tag {
    UINT  wMid;
    UINT  wPid;
    VERSION  vDriverVersion;
    char  szPname[MAXPNAMELEN];
    DWORD  dwFormats;
    UINT  wChannels;
} WAVEINCAPS;
```

Fields

The **WAVEINCAPS** structure has the following fields:

wMid

Specifies a manufacturer ID for the device driver for the waveform input device. Manufacturer IDs are defined in Appendix B, "Manufacturer ID and Product ID Lists."

wPid

Specifies a product ID for the waveform input device. Product IDs are defined in Appendix B, "Manufacturer ID and Product ID Lists."

vDriverVersion

Specifies the version number of the device driver for the waveform input device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

wChannels

Specifies whether the device supports mono (1) or stereo (2) input.

See Also **waveInGetDevCaps**

WAVEOUTCAPS

The **WAVEOUTCAPS** structure describes the capabilities of a waveform output device.

```
typedef struct waveoutcaps_tag {
    UINT  wMid;
    UINT  wPid;
    VERSION  vDriverVersion;
    char  szPname[MAXPNAMELEN];
    DWORD  dwFormats;
    UINT  wChannels;
    DWORD  dwSupport;
} WAVEOUTCAPS;
```

Fields The **WAVEOUTCAPS** structure has the following fields:

wMid

Specifies a manufacturer ID for the device driver for the waveform output device. Manufacturer IDs are defined in Appendix B, "Manufacturer ID and Product ID Lists."

wPid

Specifies a product ID for the waveform output device. Product IDs are defined in Appendix B, "Manufacturer ID and Product ID Lists."

vDriverVersion

Specifies the version number of the device driver for the waveform output device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

wChannels

Specifies whether the device supports mono (1) or stereo (2) output.

dwSupport

Specifies optional functionality supported by the device.

WAVECAPS_PITCH

Supports pitch control.

WAVECAPS_PLAYBACKRATE

Supports playback rate control.

WAVECAPS_SYNC

Specifies that the driver is synchronous and will block while playing a buffer.

WAVECAPS_VOLUME

Supports volume control.

WAVECAPS_LRVOLUME

Supports separate left and right volume control.

Comments

If a device supports volume changes, the WAVECAPS_VOLUME flag will be set for the **dwSupport** field. If a device supports separate volume changes on the left and right channels, both the WAVECAPS_VOLUME and the WAVECAPS_LRVOLUME flags will be set for this field.

See Also

waveOutGetDevCaps

Chapter 7
MCI Command Strings

The Media Control Interface (MCI) is a high-level command interface to multimedia devices and resource files. MCI provides applications with device-independent capabilities for controlling audio and visual peripherals. Your application can use MCI to control any supported multimedia device, including audio playback and recording. For a full overview of MCI, see the *Multimedia Programmer's Guide*.

MCI provides standard commands for playing multimedia devices and recording multimedia resource files. Using MCI, an application can control multimedia devices using simple commands like **open**, **play**, and **close**. MCI commands are a generic interface to multimedia devices.

MCI includes the following interfaces:

- The *command-message interface* consists of C constants and structures. The *Multimedia Programmer's Guide* describes the command-message interface and presents numerous examples on using the command messages to control audio devices. In this reference, the individual command messages and structures are described in Chapter 4, "Message Overview," Chapter 5, "Message Directory," and Chapter 6, "Data Types and Structures."
- The *command-string interface* provides a textual version of the command messages. The strings use an easy-to-read format that makes it easy to control MCI devices from C programs and multimedia authoring tools. Command strings duplicate the functionality of the command messages; Windows converts the command strings to command messages before sending them to the MCI driver for processing.

This chapter describes the command-string interface. The following topics are covered:

- MCI device types and drivers
- Driver support for MCI commands
- Syntax of MCI command strings
- Opening, playing, and closing MCI devices
- The command sets for the various MCI device types

Device Type	Description
animation	Animation device
cdaudio	audio CD player
dat	Digital audio tape player
digitalvideo	Digital video in a window (not GDI based)
other	Undefined MCI device
overlay	Overlay device (analog video in a window)
scanner	Image scanner
sequencer	MIDI sequencer
vcr	Videotape recorder or player
videodisc	Videodisc player
waveaudio	Audio device that plays digitized waveform files

In this chapter, device type names are italic.

Device Names

For any given device type, there might be several MCI drivers that share the command set but operate on different data formats. For example, the *animation* device type might include several MCI drivers that use the same command set but play different types of animation files. To uniquely identify a MCI driver, MCI uses *device names.*

Device names are identified in the [mci] section of the SYSTEM.INI file. The [mci] section of SYSTEM.INI identifies all MCI drivers to Windows. The following example shows a typical [mci] section:

```
[mci]
waveaudio=mciwave.drv
sequencer=mciseq.drv
MMMovie=mcimmp.drv
cdaudio=mcicda.drv
```

The name on the left side of the equal sign is the driver device name. The value on the right side of the equal sign identifies the filename of the MCI driver. Frequently, the device name is the same as the device type for the driver, as is the case for the *waveaudio, sequencer,* and *cdaudio* devices in the preceding example. The "MMMovie" device is an *animation* device, but it uses a unique device name.

Driver Support for MCI Commands

MCI drivers provide the functionality for MCI commands. The Windows system software performs some housekeeping tasks, but all the multimedia playback, presentation, and recording is handled by the individual MCI drivers.

Drivers vary in their support for MCI commands and command options. Because multimedia devices can have widely different capabilities, MCI is designed to let individual drivers extend or reduce the command sets to match the capabilities of the device. For example, the **record** command is part of the command set for MIDI sequencers, but the MCISEQ driver included with Windows does not support the **record** command. Also, the MCI Movie Player driver (MCIMMP.DRV) included with the Microsoft Software Development Kit extends the **open** command to include an **expanddibs** option that is not part of the basic command set for the *animation* device type.

Classifications of MCI Commands

MCI defines four classifications of commands. The commands and options comprising the following two classifications are defined as the minimum command set for any MCI driver:

- *System commands*. These commands are handled directly by MCI rather than by the driver.
- *Required commands*. These commands are handled by the driver. All drivers should support the required commands and options.

Regardless of the specific driver you're using, your application should be able to assume that the commands and options in the two preceding groups are available.

The commands comprising the following two classifications are not supported by all drivers:

- *Basic commands*, or optional commands, are used by some devices. If a device supports a basic command, it must support a defined set of options for the command.
- *Extended commands* are specific to a certain device types or drivers. Extended commands include new commands (like the **put** and **where** commands for the *animation* device type) and extensions to existing commands (like the **can stretch** option added to the *animation* **status** command).

If your application needs to use a basic or extended command or option, it should query the driver before trying to use the command or option (that is, unless you are certain that the MCI driver you've used during development is the same one that will be available on the delivery system). The following sections summarize the specific commands in each category.

Basic Commands

The following list summarizes the basic commands. The use of these messages by a device is optional.

Message	Description
load	Loads data from a file.
pause	Stops playing.
play	Starts transmitting output data.
record	Starts recording input data.
resume	Resumes playing or recording on a paused device.
save	Saves data to a disk file.
seek	Seeks forward or backward.
set	Sets the operating state of the device.
status	Obtains status information about the device. The **status** command is also listed in the group of required commands; in the basic group, options are added for devices that use linear media with identifiable positions.
stop	Stops playing.

If a driver supports a basic command, it must also support a standard set of options for the command. For a complete description of the basic commands and options, see "Basic Commands for Specific Device Types," later in this chapter.

Extended Commands

Some MCI devices have additional commands or add options to existing commands. While some extended commands only apply to a specific device driver, most of them apply to all drivers of a particular device type. For example, the *sequencer* command set extends the **set** command to add time formats that are needed by MIDI sequencers.

Unless you are certain that the specific MCI driver you use during development will be available on the delivery system, you should not assume that the device supports the extended commands or options. You can use the **capability** command to determine whether a specific feature is supported, and your application should be ready to deal with "unsupported command" or "unsupported function" return values.

For example, consider the following commands sent to the *waveaudio* driver (MCIWAVE.DRV):

```
open sound.wav alias sound
play sound notify
record sound from 0 notify
```

The **record** command returns a "Parameter out of range" value and stops the playback started by the previous **play** command. One might expect the driver to validate the **record** command before stopping playback, but the driver stops the playback first.

Using MCI Command Strings

The **mciSendString** function sends MCI command strings to MCI devices. Chapter 3, "Function Directory," describes this function in detail. Also, the *Multimedia Programmer's Guide* describes how to use **mciSendString**.

Many multimedia authoring tools let you send command strings to MCI devices. The MCITEST application, one of the sample applications included with the SDK, lets you develop test scripts that control MCI devices using command strings.

Syntax of Command Strings

MCI command strings use a consistent verb-object-modifier syntax. Each command string includes a command, a device identifier, and command arguments. Arguments are optional on some commands and required on other commands.

A command string has the following form:

command device_id arguments

These components contain the following information:

- The *command* specifies an MCI command (for example, **open**, **close**, or **play**).
- The *device_id* identifies an instance of an MCI driver. The *device_id* is created when the device is opened. See "Opening a Device," later in this chapter, for information on how the device ID is assigned.
- The *arguments* specify the flags and parameters used by the *command*. Flags are key words recognized with the MCI command. Parameters are numbers or strings that apply to the MCI command or flag.

The user can cancel a wait operation by pressing a break key. By default, this key is CTRL+BREAK. When a wait operation is cancelled, MCI attempts to return control to the application without interrupting the command associated with the **wait** flag.

For example, in the **play** command shown in the preceding example, breaking the command cancels the wait operation without interrupting the play operation.

To redefine the break key, use the **break** command.

Using the Notify Flag

The **notify** flag directs the device to post an MM_MCINOTIFY message when the device completes an action. Your application must have a window procedure to process the MM_MCINOTIFY message for notification to have any effect. The *Multimedia Programmer's Guide* includes examples of window procedures that process the MM_MCINOTIFY message.

A notification message can indicate one of the following results:

- The notification is successful
- The notification is superseded
- The notification is aborted
- The notification fails

A successful notification occurs when the conditions required for initiating the callback are satisfied and the command completed without interruption.

A notification is superseded when the device has a notification pending and you send it another notify request. When a notification is superseded, MCI resets the callback conditions to correspond to the notify request of the new command.

A notification is aborted when you send a new command that prevents the callback conditions set by a previous command from being satisfied. For example, sending the stop command cancels a notification pending for the "play to 500 notify" command. If your command interrupts a command that has a notification pending, and your command also requests notification, MCI will abort the first notification immediately and respond to the second notification.

A notification fails if a device error occurs while a device is executing the MCI command. For example, a notification fails when a hardware error occurs during a play command.

How the Device ID is Assigned

The device ID established by the open command identifies the open MCI device in all subsequent commands. If you specify a device *alias* using the **alias** key word, the device ID will be the *alias*. Otherwise, the device ID will be the *device* name.

Opening Simple Devices

MCI classifies device drivers as *compound* and *simple*. Simple device drivers don't require a device element for playback. Simple devices include *cdaudio* devices and *videodisc* devices.

For example, you can open a videodisc device using the following command:

```
open videodisc
```

Simple devices require only the *device_name* for operation. You don't need to provide any additional information (such as a name of a data file) to open these devices. For these devices, substitute the name of a device name from the [mci] section of SYSTEM.INI for the *device_name*.

Opening Compound Devices

Compound device drivers use a *device element*—a media element associated with a device—during operation. For most compound device drivers, the device element is the source or destination data file. For these file elements, the element name references a file and its path. Compound devices include *waveaudio* devices and *sequencer* devices.

Depending on your needs, there are three ways you can open a compound device:

- By specifying just the device name (this lets you open a compound device without associating an element filename). When opened this way, most compound devices will only process the **capability** and **close** commands.
- By specifying just the element name (the device name is determined from the [mci extensions] section of the WIN.INI file).
- By specifying both the element name and the device name (MCI ignores the entries in the [mci extensions] section of the WIN.INI file and opens the specified device name).

To associate a device element with a particular device, you can specify the element name and device name. For example, the following command opens the *waveaudio* device with element filename "MYVOICE.SND":

```
open myvoice.snd type waveaudio
```

Opening Shareable Devices

The **shareable** flag lets multiple applications access the same device (or element) and device instance concurrently. If your application opens a device or device element as shareable, other applications can also access it by opening it as shareable. The shared device or device element gives each application the ability to change the parameters governing the operating state of the device or device element. Each time a device or device element is opened as shareable, MCI returns a unique device ID even though the IDs refer to the same instance.

If your application opens a device or device element without the **shareable** flag, no other application can access it simultaneously. Also, if a device can service only one open instance, the **open** command will fail if you specify the **shareable** flag.

If you make a device or device element shareable, your application should not make any assumptions about the state of a device. When working with shared devices, your application might need to compensate for changes made by other applications using the same services.

While most compound device elements are not shareable, you can open multiple elements (where each element is unique), or you can open a single element multiple times. If you open a single file element multiple times, MCI creates an independent instance for each, with each instance having a unique operating status.

If you open multiple instances of a file element, you must assign a unique device ID to each. The **alias** flag described in the following section lets you assign a unique name for each element.

Assigning a Device ID Using the Alias Flag

The **alias** flag specifies a device ID for the device. The alias flag lets you assign a short device ID for compound devices with lengthy filenames, and it lets you open multiple instances of the same file or device.

For example, the following command assigns device ID "birdcall" to a lengthy waveform filename:

```
open c:\nabirds\sounds\mockmtng.wav type waveaudio alias birdcall
```

If the alias flag were omitted, the device ID would be "c:\nabirds\sounds\mockmtng.wav."

Stopping and Pausing a Device

The **stop** command suspends the playing or recording of a device. Many devices also support the **pause** command. The difference between **stop** and **pause** depends on the device. Usually **pause** suspends operation but leaves the device ready to resume playing or recording immediately.

Using **play** or **record** to restart a device will reset the **to** and **from** positions specified before the device was paused or stopped. Without the **from** flag, these commands reset the start position to the current position. Without the **to** flag, they reset the end position to the end of the media.

To continue playing or recording while stopping at a previously specified position, use the **to** flag with the **play** or **record** commands to specify an ending position.

Some devices include the **resume** command to restart a paused device. This command does not change the **to** and **from** positions specified with the **play** or **record** command which preceded the **pause** command.

Closing a Device

The **close** command releases access to a device or device element. To help MCI manage the devices, your application must explicitly close each device or device element when it is finished with it.

Shortcuts and Variations for MCI Commands

The MCI command-string interface lets you use several shortcuts when working with MCI devices.

Using All as a Device ID

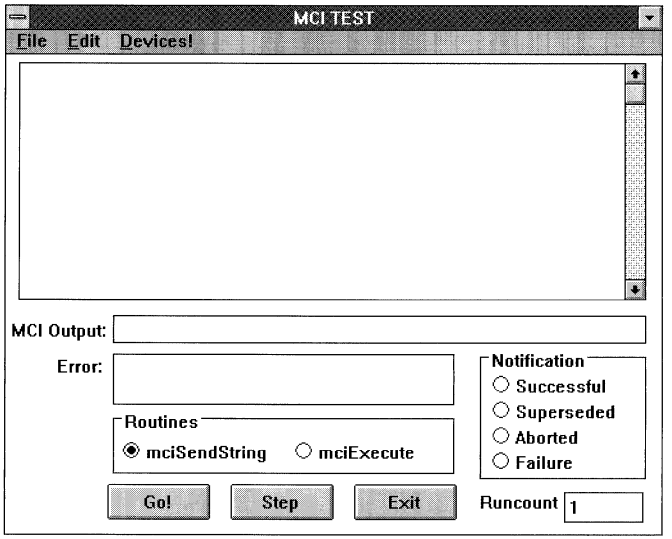
You can specify **all** as a device ID for any command that does not return information. When you specify **all**, MCI sequentially sends the command to all devices opened by the current application.

For example, "close all" closes all open devices and "play all" starts playing all devices opened by the application. Because MCI sequentially sends the commands to the MCI devices, there is a delay between when the first device receives the command and when the last device receives the command.

Using the MCITEST Application

The MCITEST sample application on your SDK disks provides a simple way to experiment with MCI command strings. It lets you type MCI command strings and send them to Windows for processing.

When you run MCITEST, it displays the following dialog box:



To try an MCI command, enter the command string in the large edit box. MCITEST sends the string directly to the MCI command-string interface when you press ENTER. Any MCI response to the command is displayed in the MCI Output box. Any errors returned by MCI appear in the Error box.

The Routines options select the function used to send the command strings to MCI.

The GO button sequentially sends all commands in the edit box to the MCI command-string interface. You can have MCITEST execute the entire command list multiple times by specifying a number in the Runcount box.

The STEP button sends the selected MCI command, then moves the selection to the next command. The EXIT button ends MCITEST.

MCI sets the option buttons in the Notification are in response to the notify flag.

The File menu displays commands you can use to save and recall the contents of the edit box. The Edit menu displays commands you can use to edit the contents of the edit box. The Device menu displays a of open device instances.

To hear the song at a different speed, use the following command before playing the song:

```
set song tempo 200
```

When you are finished using MCITEST, close all the MCI devices you opened before exiting. For example, the following command closes all the devices you opened during the session:

```
close all
```

Closing an application with open MCI devices can prevent other applications from using those devices until Windows is restarted.

MCI System Commands

The following commands are interpreted directly by MCI. The remaining command tables list the commands interpreted by the devices.

break

Syntax

break *device_id parameter* [**notify**] [**wait**]

The **break** command specifies a key to abort a **wait** command.

Parameters

Specify one of the following items for *parameter*:

on *virtual_key*

Specifies the *virtual_key* code that aborts the **wait**. When the key is pressed, the device returns control to the application. If possible, the command continues execution. Substitute a Windows virtual key code for *virtual_key*.

off

Disables the current break key.

Example

The following command sets F2 as the break key for the "mysound" device:

```
break mysound on 113
```

Required Commands for All Devices

The following required commands are recognized by all devices. Extended commands can add other options to these commands.

capability

Syntax

capability *device_id parameter* [**notify**] [**wait**]

The **capability** command requests information about a particular capability of a device.

Parameters

Specify one of the following items for *parameter* (specific device types and drivers may define other items):

can eject
Returns **true** if the device can eject the media.

can play
Returns **true** if the device can play.

can record
Returns **true** if the device supports recording.

can save
Returns **true** if the device can save data.

compound device
Returns **true** if the device supports an element name.

device type
Returns one of the following device type names:

animation
cdaudio
dat
digitalvideo
other
overlay
scanner
sequencer
vcr
videodisc
waveaudio

has audio
Returns **true** if the device supports audio playback.

Example The following command gets a description of the hardware associated with the "mysound" device:

```
info mysound product
```

open

Syntax **open** *device_id* [*parameters*] [**notify**] [**wait**]

The **open** command initializes the device.

Parameters You can specify one or more of the following optional items for *parameters* (specific device types and drivers may define other items):

alias *device_alias*

Specifies an alternate name for the given device. If specified, it must be used as the *device_id* in subsequent commands.

shareable

Initializes the device or element as shareable. Subsequent attempts to open it fail unless you specify **shareable** in both the original and later **open** commands. MCI returns an invalid device error if the device is already open and not shareable.

type *device_type*

Specifies the device type of a device element. As an alternative to **type**, MCI can use the [mci extension] entries in the SYSTEM.INI file to select the device based on the extension used by the device element. You can also use the *device_name*!*element_name* abbreviation described in "Combining the Device Name and Element Name," earlier in this chapter.

Example The following command opens the *cdaudio* device and assigns an alias:

```
open cdaudio alias cd
```

Parameters You can specify the following optional parameter:

filename
Specifies the source path and file.

Example The following command loads a file into the "vidboard" device:

```
load vidboard c:\vid\fish.vid notify
```

The **notify** flag tells MCI to send a notification message when the loading completes.

pause

Syntax **pause** *device_id* [**notify**] [**wait**]

The **pause** command pauses playing or recording. Most drivers retain the current position, allowing playback or recording to continue at the current position.

Example The following command pauses the "mysound" device:

```
pause mysound
```

play

Syntax **play** *device_id* [*parameters*] [**notify**] [**wait**]

The **play** command starts playing the device.

Parameters You can specify one or more of the following optional items for *parameters*:

from *position*
Specifies a starting position for the playback. If the **from** parameter is not specified, playback begins at the current position.

to *position*
Specifies an ending position for the playback. If the **to** parameter is not specified, playback ends at the end of the media.

Comments Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example The following command plays the mysound device from position 1000 through position 2000, sending a notification message when the playback completes:

```
play mysound from 1000 to 2000 notify
```

save

Syntax

save *device_id* [*filename*] [**notify**] [**wait**]

The **save** command saves the MCI element.

Parameters

You can specify the following optional item:

filename
Specifies the destination path and file.

Comments

The *filename* parameter is required if the device was opened using the **new** device ID.

Example

The following command saves the data in the "newsound" device to C:\SOUNDS\NEWSND.WAV:

```
save newsound c:\sounds\newsnd.wav
```

seek

Syntax

seek *device_id parameter* [**notify**] [**wait**]

The **seek** command moves to the specified position and stops.

Parameters

Specify one of the following items for *parameter*:

to *position*
Specifies the position to stop the seek.

to start
Seeks to the start of the media.

to end
Seeks to the end of the media.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command seeks to the start of the media file associated with the "mysound" device:

```
seek mysound to start
```

status

Syntax

status *device_id parameter* [**notify**] [**wait**]

The **status** command gets status information for the device. This command is also listed as a required command. As a basic command, **status** adds flags for devices with linear media.

Parameters

Specify one of the following items for *parameter*:

current track
Returns the current track.

length
Returns the total length of the media.

length track *track_number*
Returns the length of the track specified by *track_number*.

number of tracks
Returns the number of tracks on the media.

position
Returns the current position.

position track *track_number*
Returns the position of the start of the track specified by *track_number*.

ready
Returns **true** if the device is ready to play.

start position
Returns the starting position of the media.

time format
Returns the current time format.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command returns the time format used by the "mysound" device:

```
status mysound time format
```

device type
Animation devices return **animation**.

fast play rate
Returns fast play rate in frames per second.

has audio
Returns **true** if the device supports audio playback.

has video
Returns **true**.

normal play rate
Returns normal play rate in frames per second.

slow play rate
Returns the slow play rate in frames per second.

uses files
Returns **true** if the element of a compound device is a file path.

uses palettes
Returns **true** if the device uses palettes.

windows
Returns the number of windows the device can support.

Example

The following command returns the fast play rate of the "movie" device:

```
capability movie fast play rate
```

close

Syntax

close *device_id* [**notify**] [**wait**]

The **close** command closes a device element and any associated resources.

Example

The following command closes the "movie" device:

```
close movie
```

style *style_type*
Indicates a window style.

style child
Opens a window with a child window style.

style overlapped
Opens a window with an overlapped window style.

style popup
Opens a window with a popup window style.

type *device_type*
Specifies the device type of the device element. As an alternative to **type**, MCI can use the [mci extension] entries in the SYSTEM.INI file to select the controlling device based on the extension used by the device element. You can also use the *device_name*!*element_name* abbreviation described in "Combining the Device Name and Element Name," earlier in this chapter.

Example

The following command opens animation file \MMM\SNQLMFLL.MMM:

```
open \mmm\snqlmfll.mmm type mmmovie alias movie style 13369344
```

The command specifies device type "mmmovie," assigns an alias of "movie," and specifies an overlapped window with a system menu, caption, thick border, and maximize box (combination of the window styles WS_OVERLAPPED, WS_SYSMENU, WS_BORDER, WS_THICKFRAME, and WS_MAXIMIZEBOX).

pause

Syntax

pause *device_id* [**notify**] [**wait**]

The **pause** command pauses playback of the animation. If the animation is stopped, **pause** displays the animation window if it is not already visible and in the foreground.

Example

The following command pauses the "movie" device:

```
pause movie
```

put

Syntax

put *device_id parameter* **[notify] [wait]**

The **put** command defines the area of the source image and destination window used for display.

Parameters

Specify one of the following items for *parameter*:

destination

Sets the whole window as the destination window.

destination at *rectangle*

Specifies a rectangle for the area of the window used to display the image. The *rectangle* coordinates are relative to the window origin and are specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the top-left corner, and the coordinates *X2 Y2* specify the width and height of the rectangle.

When an area of the display window is specified, and the device supports stretching, the source image is stretched to the destination offset and extent.

source

Selects the whole image for display in the destination window.

source at *rectangle*

Specifies a rectangle for the image area used for display. The *rectangle* coordinates are relative to the image origin and are specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the top-left corner, and the coordinates *X2 Y2* specify the width and height of the rectangle.

When an area of the source image is specified, and the device supports stretching, the source image is stretched to the destination offset and extent.

Example

The following command sets the display area for the movie device. It specifies an offset of 10 pixels from the top-left corner of the playback window. The playback area is clipped to 250 pixels wide and 340 pixels high.

```
put movie destination at 10 10 250 340
```

Example

The following command seeks to the beginning of the animation file associated with the "movie" device:

```
seek movie to start
```

set

Syntax

set *device_id parameters* [**notify**] [**wait**]

The **set** command establishes control settings for the driver.

Parameters

Specify one of the following items for *parameters*:

audio all off
Disables audio output.

audio all on
Enables audio output.

audio left off
Disables output to the left audio channel.

audio left on
Enables output to the left audio channel.

audio right off
Disables output to the right audio channel.

audio right on
Enables output to the right audio channel.

time format frames
Sets the time format to frames. All commands that use position values will assume frames. When the device is opened, frames is the default mode.

time format milliseconds
Sets the time format to milliseconds. All commands that use position values will assume milliseconds. You can abbreviate milliseconds as **ms**.

number of tracks
Returns the number of tracks on the media.

palette handle
Returns the handle of the palette used for the animation.

position
Returns the current position.

position track *number*
Returns the position of the start of the track specified by *number.*

ready
Returns **true** if the device is ready.

speed
Returns the current speed of the device in frames per second.

start position
Returns the starting position of the media.

stretch
Returns **true** if stretching is enabled.

time format
Returns the current time format.

window handle
Returns the handle of the window used for the animation.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command a handle to the current palette used by the "movie" device:

```
status movie palette handle
```

Example The following command updates the entire display window used by the "movie" device. The number following **hdc** is a handle to a device context obtained from the **BeginPaint** function:

```
update movie hdc 203
```

where

Syntax **where** *device_id parameter* [**notify**] [**wait**]

The **where** command gets the rectangle specifying the source or destination area.

Parameters Specify one of the following items for *parameter*:

destination
Requests the destination offset and extent.

source
Requests the source offset and extent.

Example The following command returns the display rectangle of the "movie" device:

```
where movie destination
```

window

Syntax **window** *device_id parameters* [**notify**] [**wait**]

The **window** command controls the animation display window. You can change the display characteristics of the window or provide a display window for the driver to use in place of the default display window.

Generally, animation devices create a window when opened but don't display the window until they receive a **play** command. If your application provides a window to the driver, your application is responsible for managing the messages sent to the window.

The window command provides several flags that let you manipulate the window. Since you can use the **status** command to get the handle to the driver display window, you can also use the standard window manager functions (like **ShowWindow**) to manipulate the window.

Example The following command displays and sets the caption for the "movie" playback window:

```
window movie text "Welcome to the Movies" state show
```

Audio CD (Red Book) Commands

The *cdaudio* command set provides a common method for playing audio CD. This device type includes the MCICDA.DRV device driver, which operates with any CD-ROM device supporting the audio services of MSCDEX. Audio CD devices support the following set of commands:

capability

Syntax **capability** *device_id parameter* [**notify**] [**wait**]

The **capability** command requests information about audio CD capabilities of the device.

Parameters Specify one of the following items for *parameter*:

can eject
Returns **true** if the audio CD device can eject the media.

can play
Returns **true** if the audio CD device can play the media.

can record
Returns **false**. audio CD devices cannot record.

can save
Returns **false**. audio CD devices cannot save data.

compound device
Returns **false**. audio CD devices are simple devices.

device type
Returns **cdaudio**.

has audio
Returns **true**.

has video
Returns **false**. audio CD devices don't support video.

uses files
Returns **false**. Simple devices don't use files.

Example The following command opens the *cdaudio* device:

```
open cdaudio
```

pause

Syntax **pause** *device_id* [**notify**] [**wait**]

The **pause** command pauses playing. With the MCICDA driver, the **pause** command works the same as the **stop** command.

Example The following command pauses the *cdaudio* device:

```
pause cdaudio
```

play

Syntax **play** *device_id* [*parameters*] [**notify**] [**wait**]

The **play** command starts playing the audio disc.

Parameters You can specify one or more of the following optional items for *parameters*:

from *position*

Specifies the position to start playback. If the **from** parameter is not specified, playback begins at the current position. If the **from** position is greater than the end position of the disc, or if the **from** position is greater than the **to** position, the driver returns an error.

to *position*

Specifies the position to stop playback. By default, playback continues to the end of the disc. If the **to** position is greater than the length of the disc, MCI returns an error.

Comments Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example The following command plays tracks two through five of the audio disc (assuming the time format is set to TMSF):

```
play cdaudio from 2 to 6
```

Parameters

Specify one of the following items for *parameters*:

audio all off
Disables audio output.

audio all on
Enables audio output.

audio left off
Disables output to the left audio channel.

audio left on
Enables output to the left audio channel.

audio right off
Disables output to the right audio channel.

audio right on
Enables output to the right audio channel.

door closed
Retracts the tray and closes the door if possible.

door open
Opens the door and ejects the tray if possible.

time format milliseconds
Sets the time format to milliseconds. All commands that use position values will assume milliseconds. You can abbreviate milliseconds as **ms**.

time format msf
Sets the time format to minutes, seconds, and frames. All commands that use position values will assume MSF (the default format for audio CD).

Specify an MSF value as *mm*:*ss*:*ff*, where *mm* is minutes, *ss* is seconds, and *ff* is frames. You can omit a field if it and all following fields are zero. For example, 3, 3:0, and 3:0:0 are valid ways to express 3 minutes.

The MSF fields have the following maximum values:

Minutes 99
Seconds 59
Frames 74

mode
Returns one of the following values indicating the current mode of the device:

not ready
open
paused
playing
seeking
stopped

number of tracks
Returns the number of tracks on the disc.

position
Returns the current position.

position track *track_number*
Returns the starting position of the track specified by *track_number*.

ready
Returns **true** if the device is ready.

start position
Returns the starting position of the disc or device element.

time format
Returns the current time format.

Comments Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example The following command returns the number of tracks on the current disc:

```
status cdaudio number of tracks
```

stop

Syntax **stop** *device_id* [**notify**] [**wait**]

The **stop** command stops playback.

Example The following command stops the *cdaudio* device:

```
stop cdaudio
```

has audio
Returns **true**. Sequencers support playback.

has video
Returns **false**. Sequencers don't support video.

uses files
Returns **true**. Sequencers use files for operation.

Example The following command returns true if the "music" device can record:

```
capability music can record
```

close

Syntax **close** *device_id* [**notify**] [**wait**]

The **close** command closes the sequencer device, as well as the associated port and file.

Example The following command closes the "music" device:

```
close music
```

info

Syntax **info** *device_id parameter* [**notify**] [**wait**]

The **info** command gets textual information from the device.

Parameters Specify one of the following items for *parameter*:

file
Returns the filename of the current MIDI file.

product
Returns the product name of the sequencer. The MCISEQ sequencer returns **MIDI Sequencer**.

Example The following command returns the filename of the MIDI file associated with the "music" device:

```
info music file
```

play

Syntax

play *device_id* [*parameters*] [**notify**] [**wait**]

The **play** command starts playing the sequencer.

MIDI files played using the MCISEQ sequencer should be authored to conform to the Microsoft MIDI file authoring guidelines, which are discussed in the *Multimedia Programmer's Guide*. If the file does not conform, the MCISEQ sequencer displays a warning dialog box when you issue the **play** command for the file.

Parameters

You can specify one or more of the following optional items for *parameters*:

from *position*

Specifies a starting position for the playback. If the **from** parameter is not specified, playback begins at the current position.

to *position*

Specifies an ending position for the playback. If the **to** parameter is not specified, playback ends at the end of the media.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command starts playing the "clavier" device from the start of the MIDI file:

```
play clavier from 0
```

record

Syntax

record *device_id* [*parameters*] [**notify**] [**wait**]

The **record** command starts recording MIDI data. All data recorded after a file is opened is discarded if the file is closed without saving it. The MCISEQ sequencer does not support recording.

save

Syntax

save *device_id* [*filename*] [**notify**] [**wait**]

The **save** command saves the MCI element. The MCISEQ sequencer does not support this command.

Parameters

You can specify the following optional item:

filename
The *filename* specifies the destination path and file.

Comments

The *filename* parameter is required if the device was opened using the **new** device ID.

Example

The following command saves the MIDI data recorded into the "clavier" device as filename C:\MIDI\MYMIDI.MID:

```
save clavier c:\midi\mymidi.mid
```

seek

Syntax

seek *device_id parameter* [**notify**] [**wait**]

The **seek** command moves to the specified position in the file.

Parameters

Specify one of the following items for *parameter*:

to *position*
Specifies to seek to *position.*

to start
Seeks to the start of the sequence.

to end
Seeks to the end of the sequence.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command moves to the beginning of the MIDI file associated with the "clavier" device:

```
seek clavier to start
```

port *port_number*

Sets the MIDI port receiving the MIDI messages. This command will fail if the port you are trying to open is being used by another application.

port mapper

Sets the MIDI mapper as the port receiving the MIDI messages. This command will fail if the MIDI mapper or a port it needs is being used by another application.

port none

Disables the sending of MIDI messages. This command also closes a MIDI port.

slave file

Sets the MIDI sequencer to use file data as the synchronization source. This is the default.

slave MIDI

Sets the MIDI sequencer to use incoming data MIDI for the synchronization source. The sequencer recognizes synchronization data with the MIDI format. The MCISEQ sequencer does not support this option.

slave none

Sets the MIDI sequencer to ignore synchronization data.

slave SMPTE

Sets the MIDI sequencer to use incoming MIDI data for the synchronization source. The sequencer recognizes synchronization data with the SMPTE format. The MCISEQ sequencer does not support this option.

tempo *tempo_value*

Sets the tempo of the sequence according to the current time format. For a ppqn-based file, the *tempo_value* is interpreted as beats per minute. For a SMPTE-based file, the *tempo_value* is interpreted as frames per second.

time format milliseconds

Sets the time format to milliseconds. All commands that use position values will assume milliseconds. You can abbreviate milliseconds as **ms**.

The sequence file sets the default format to ppqn or SMPTE.

time format song pointer

Sets the time format to song pointer (sixteenth notes). All commands that use position values will assume song pointer units. This option is valid only for a sequence of division type ppqn.

length

Returns the length of a sequence in the current time format. For ppqn files, this will be song pointer units. For SMPTE files, this will be expressed as *hh:mm:ss:ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.

length track *track_number*

Returns the length of the sequence in the current time format. For ppqn files, this will be song pointer units. For SMPTE files, this will be expressed as *hh:mm:ss:ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.

master

Returns **midi**, **none**, or **smpte** depending on the type of synchronization set.

media present

The sequencer returns **true**.

mode

Returns **not ready**, **paused**, **playing**, **seeking**, or **stopped**.

number of tracks

Returns the number of tracks. MCISEQ returns 1.

offset

Returns the offset of a SMPTE-based file. The offset is the start time of a SMPTE based sequence. The time is returned as *hh:mm:ss:ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.

port

Returns the MIDI port number assigned to the sequence.

position

Returns the current position of a sequence in the current time format. For ppqn files, this will be song pointer units. For SMPTE files, this will be in form *hh:mm:ss:ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.

position track *track_number*

Returns the current position of the track specified by *track_number* in the current time format. For ppqn files, this will be song pointer units. For SMPTE files, this will be in form *hh:mm:ss:ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames. The MCISEQ sequencer returns 0.

ready

Returns **true** if the device is ready.

slave

Returns **file**, **midi**, **none**, or **smpte** depending on the type of synchronization set.

Videodisc Player Commands

The *videodisc* command set provides a common method for playing videodiscs. This device type includes the MCIPIONR.DRV driver, which operates with the Pioneer LD-V4200 videodisc player. Videodisc players support the following set of commands:

capability

Syntax

capability *device_id parameter* [**notify**] [**wait**]

The **capability** command requests information about a particular capability of a device. The return information is for the type of disc inserted unless the **CAV** or **CLV** options are used to override the format. If no disc is present, information is returned for **CAV** discs.

Parameters

Specify one of the following items for *parameter*:

can eject
Returns **true** if the device can eject the disc. The MCIPIONR device returns **true**.

can play
Returns **true** if the device supports playing. The MCIPIONR device returns **true**.

can record
Returns **true** if the video device can record. The MCIPIONR device returns **false**.

can reverse
Returns **true** if the device can play in reverse, **false** otherwise. CLV discs return **false**.

can save
Returns **false**. MCI videodisc players cannot save data.

CAV
When combined with other items, **CAV** specifies that the return information applies to CAV format discs. This is the default if no disc is inserted.

CLV
When combined with other items, **CLV** specifies that the return information applies to CLV format discs.

compound device
Returns **false**. MCI videodisc players are simple devices.

device type
Returns **videodisc**.

Example The following command sends the escape string "SA" to the *videodisc* device:

```
escape videodisc SA
```

info

Syntax **info** *device_id parameter* [**notify**] [**wait**]

The **info** command obtains textual information from a device.

Parameters Specify the following item for *parameter*:

product
Returns the product name of the device that the driver is controlling. The MCIPIONR device returns **Pioneer LD-V4200.**

Example The following command returns the product name of the device controlled by the *videodisc* device:

```
info videodisc product
```

open

Syntax **open** *device_id* [*parameters*] [**notify**] [**wait**]

The **open** command initializes the device. MCI reserves *videodisc* for the videodisc device type.

Parameters You can specify one or more of the following items for *parameters*:

alias *device_alias*
Specifies an alternate name for the given device. If specified, it must be used as the *device_id* in subsequent commands.

shareable
Initializes the device as shareable. Subsequent attempts to open it fail unless you specify **shareable** in both the original and subsequent **open** commands. MCI returns an invalid device error if the device is already open and cannot be shared.

Example The following command opens the *videodisc* device:

```
open videodisc
```

scan

Indicates the device should play as fast as possible, possibly with audio disabled. This flag applies only to CAV discs.

speed *integer*

Specifies the rate of play in frames per second (for example, speed 15 means 15 frames per second). This applies only to CAV discs.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command starts fast playback:

```
play videodisc scan
```

resume

Syntax

resume *device_id* [**notify**] [**wait**]

The **resume** command continues playing or recording on a paused device. The MCIPIONR driver does not support this command.

Example

The following command continues playback on the *videodisc* device:

```
resume videodisc
```

seek

Syntax

seek *device_id* [*parameter*] [**notify**] [**wait**]

The **seek** command searches using fast forward or fast reverse with video and audio off.

Parameters

You can specify one of the following optional items for *parameter*:

reverse

Indicates the seek direction on CAV discs is backwards. This modifier is invalid if **to** is specified.

to *position*

Specifies the position to stop the seek. If **to** is not specified, the seek continues to the end of the disc.

to start

Seeks to the start of the disc.

to end

Seeks to the end of the disc.

time format hms

Sets the time format to hours, minutes, and seconds. All commands that use position values will assume HMS. HMS is the default format for CLV discs.

Specify an HMS value as *hh:mm:ss*, where *hh* is hours, *mm* is minutes, and *ss* is seconds. You can omit a field if it and all following fields are zero. For example, 3, 3:0, and 3:0:0 are all valid ways to express 3 hours.

time format milliseconds

Sets the time format to milliseconds. All commands that use position values will assume milliseconds. You can abbreviate milliseconds as **ms**.

time format track

Sets the position format to tracks. All commands that use position values will assume tracks.

video off

Disables video output.

video on

Enables video output.

Example

The following command closes the door of the device and sets the time format to milliseconds:

```
set videodisc time format ms door closed
```

spin

Syntax

spin *device_id parameter* [**notify**] [**wait**]

The **spin** command starts the disc spinning or stops the disc from spinning.

Parameters

Specify one of the following items for *parameter*:

down

Stops the disc from spinning.

up

Starts the disc spinning.

Example

The following command spins up the *videodisc* device:

```
spin videodisc up
```

side
Returns 1 or 2 to indicate which side of the disc is loaded.

speed
Returns the current speed in frames per second. The MCIPIONR videodisc player does not support this option.

start position
Returns the starting position of the disc.

time format
Returns the current time format.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command returns **true** if a videodisc is inserted in the device controlled by the *videodisc* driver:

```
status videodisc media present
```

step

Syntax

step *device_id* [*parameter*] [**notify**] [**wait**]

The **step** command steps the play one or more frames forward or reverse. The default action is to step forward one frame. The **step** command applies only to CAV discs.

You can specify one or both of the following optional items for *parameter*:

by *frames*
Specifies the number of *frames* to step. If you specify a negative *frames* value, you cannot specify the **reverse** flag.

reverse
Step backward.

Example

The following command steps the videodisc one frame forward:

```
step videodisc
```

device type
Returns **overlay**.

has audio
Returns **true** if the device supports audio playback.

has video
Returns **true**. Video overlay devices are video devices.

uses files
Returns **true** if elements of the device are filenames.

windows
Returns the number of simultaneous display windows the device can support.

Example

The following command returns **true** if the overlay device supports stretching:

```
capability vboard can stretch
```

close

Syntax

close *device_id* [**notify**] [**wait**]

The **close** command closes a video overlay element and any associated resources.

Example

The following command closes the "vboard" device:

```
close vboard
```

freeze

Syntax

freeze *device_id* [*parameter*] [**notify**] [**wait**]

The **freeze** command disables video acquisition to the frame buffer. This is supported only if **capability can freeze** returns **true**.

Parameters

You can specify the following optional item for *parameter*:

at *rectangle*
Specifies the rectangular region that will have video acquisition disabled. To specify irregular acquisition regions, use a series of **freeze** and **unfreeze** commands. Some video overlay devices limit the complexity of the acquisition region.

The *rectangle* region is relative to the video buffer origin and is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the top-left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

Example

The following command loads the video capture file C:\VCAP\VCAPFILE.TGA into the video buffer:

```
load vboard c:\vcap\vcapfile.tga notify
```

The "vboard" device sends a notification message when the loading is completed.

open

Syntax

open *device_id* [*parameters*] [**notify**] [**wait**]

The **open** command initializes the video overlay device.

Parameters

You can specify one or more of the following optional items for *parameters*:

alias *device_alias*
Specifies an alternate name for the device element. If specified, it must be used as the *device_id* in subsequent commands.

parent *hwnd*
Specifies the window handle of the parent window.

shareable
Initializes the device element as shareable. Subsequent attempts to open it fail unless you specify **shareable** in both the original and subsequent **open** commands. MCI returns an error if the device is already open and cannot be shared.

style *style_type*
Indicates a window style.

style child
Opens a window with a child window style.

style overlapped
Opens a window with an overlapped window style.

style popup
Opens a window with a popup window style.

type *device_type*
Specifies the device type of the device element. MCI reserves overlay for the video overlay device type. As an alternative to **type**, MCI can use the [mci extension] entries in the SYSTEM.INI file to select the controlling device based on the extension used by the device element. You can also use the *device_name*!*element_name* abbreviation described in "Combining the Device Name and Element Name," earlier in this chapter.

frame at *rectangle*

Selects a portion of the frame buffer to receive the incoming video images. The *rectangle* coordinates are relative to the video buffer origin and are specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the top-left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

source

Selects the entire video buffer to display in the destination window.

source at *rectangle*

Selects a portion of the video buffer to display in the destination window. The *rectangle* coordinates are relative to the video buffer origin and are specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the top-left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

destination

Selects the entire client area of the destination window to display the video data from the frame buffer.

destination at *rectangle*

Selects a portion of the client area of the destination window to display the video data from the frame buffer. The *rectangle* coordinates are relative to the window origin and are specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the top-left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

Example

The following command defines three regions for the video, frame, and source:

```
put vboard video 120 120 200 200 frame 0 0 200 200 source 0 0 200 200
```

The regions are defined as follows:

- A 200- by 200-pixel region of the incoming video data, starting at an origin 120 pixels from the top-left corner, will be captured to the frame buffer.
- The video data will be placed in a 200- by 200-pixel region at the top-left corner of the frame buffer.
- Transfers are made from the 200- by 200-pixel region at the top-left corner of the frame buffer to the destination window.

time format milliseconds
Video overlay devices don't support this option.

video off
Disables video output.

video on
Enables video output.

Example The following command enables video output on the "vboard" device:

```
set vboard video on
```

status

Syntax **status** *device_id parameter* [**notify**] [**wait**]

The **status** command gets status information for the device.

Parameters Specify one of the following items for *parameter*:

media present
Returns **true**.

mode
Returns **not ready**, **recording**, or **stopped** for the current mode.

ready
Returns **true** if the video overlay device is ready.

stretch
Returns **true** if stretching is enabled.

window handle
Returns the handle of the window used for the video overlay display in the low word of the return value.

Example The following command returns true if stretching is enabled on the "vboard" device:

```
status vboard stretch
```

window

Syntax

window *device_id parameters* [**notify**] [**wait**]

The **window** command controls the destination window. The destination window is the window in which the image is displayed. You can change the display characteristics of the window or provide a destination window for the driver to use in place of the default destination window.

Generally, video overlay devices should create and display a window when opened. If your application provides a window to the driver, your application is responsible for managing the messages sent to the window.

The window command provides several flags that let you manipulate the window. Since you can use the **status** command to get the handle to the destination window, you can also use the standard window manager functions (like **ShowWindow**) to manipulate the window.

Parameters

Specify one or more of the following items for *parameters*:

fixed
Disables stretching of the image.

handle *window_handle*
Specifies the handle of a window to use instead of the default destination window.

handle default
Specifies that the video overlay device should create and manage its own destination window. This flag can be used to set the display back to the driver's default window.

state hide
Hides the destination window.

state iconic
Displays the destination window as an icon.

state maximized
Maximizes the destination window.

state minimize
Minimizes the destination window and activates the top-level window in the window-manager's list.

state minimized
Minimizes the destination window.

Parameters

Specify one of the following items for *parameter*:

can eject
Returns **false**. Waveform audio devices have no media to eject.

can play
Returns **true** if the device can play. The device returns **true** if an output device is available.

can record
Returns **true** if the device can record.

can save
Returns **true** if the device can save data.

compound device
Returns **true**; waveform audio devices are compound devices.

device type
Returns **waveaudio**.

has audio
Returns **true**.

has video
Returns **false**. Waveform audio devices don't support video.

inputs
Returns the total number of input devices.

outputs
Returns the total number of output devices.

uses files
Returns **true**. Waveform audio devices use files for operation.

Example

The following command returns the number of waveform output devices:

```
capability mysound outputs
```

Comments Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example The following command deletes the waveform data from one millisecond through 900 milliseconds (assuming the time format is set to milliseconds):

```
delete mysound from 1 to 900
```

info

Syntax **info** *device_id parameter* [**notify**] [**wait**]

The **info** command gets textual information from the device.

Parameters Specify one of the following items for *parameter*:

input
Returns the description of the current waveform audio input device. Returns **none** if an input device is not set. The MCIWAVE driver returns **Wave Audio Input and Output Device.**

file
Returns the current filename.

output
Returns the description of the current waveform audio output device. Returns **none** if an output device is not set. The MCIWAVE driver returns **Wave Audio Input and Output Device.**

product
Returns the description of the current waveform audio output device. The MCIWAVE driver returns **Wave Audio Input and Output Device.**

Example The following command returns the filename of the waveform file associated with the "mysound" device:

```
info mysound file
```

pause

Syntax

pause *device_id*

The **pause** command pauses playing or recording.

Example

The following command pauses playback or recording in the "mysound" device:

```
pause mysound
```

play

Syntax

play *device_id* [*parameters*] [**notify**] [**wait**]

The **play** command starts playing audio.

Parameters

You can specify one or more of the following optional items for *parameters*:

from *position*
: Specifies the starting position for the playback. If the **from** parameter is not specified, playback begins at the current position.

to *position*
: Specifies the ending position for the playback. If the **to** parameter is not specified, play stops at the end of the media.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command plays the "mysound" device from the beginning:

```
play mysound from 1
```

save

Syntax

save *device_id* [*filename*] [**notify**] [**wait**]

The **save** command saves the MCI element in its current format.

Parameters

You can specify the following optional item:

filename
Specifies the file and path used to save data.

Comments

The *filename* parameter is required if the device was opened using the **new** device ID.

Example

The following command saves the waveform data recorded into the "mysound" device:

```
save mysound c:\sounds\mysound.wav
```

seek

Syntax

seek *device_id parameter* [**notify**] [**wait**]

The **seek** command moves to the specified position and stops. Specify one of the following items for *parameter*:

to *position*
Specifies the stop position.

to start
Seeks to the start of the first sample.

to end
Seeks to the end of the last sample.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command seeks to the end of the waveform data in "mysound":

```
seek mysound to end
```

bytespersec *byte_rate*

Sets the average number of bytes per second played or recorded. The file is saved in this format.

channels *channel_count*

Sets the channels for playing and recording. The file is saved in this format.

format tag *tag*

Sets the format type for playing and recording. The file is saved in this format.

format tag pcm

Sets the format type to PCM for playing and recording. The file is saved in this format.

input *integer*

Sets the audio channel used as the input.

output *integer*

Sets the audio channel used as the output.

samplespersec *integer*

Sets the sample rate for playing and recording. The file is saved in this format.

time format bytes

In a PCM file format, sets the time format to bytes. All position information is specified as bytes following this command.

time format milliseconds

Sets the time format to milliseconds. All commands that use position values will assume milliseconds. You can abbreviate milliseconds as **ms**.

time format samples

Sets the time format to samples. All position information is specified as samples following this command.

Example

The following command sets the time format to milliseconds and sets the waveform audio format to 8 bit, mono, 11 kHz:

```
set mysound time format ms bitspersample 8 channels 1 samplespersec 11025
```

number of tracks
Returns the number of tracks. The MCIWAVE device returns 1.

output
Returns the currently set output. If no output is set, the error returned indicates that any device can be used.

position
Returns the current position.

position track *track_number*
Returns the position of the track specified by *track_number*. The MCIWAVE device returns 0.

ready
Returns **true** if the device is ready.

samplespersec
Returns the number of samples per second played or recorded.

start position
Returns the starting position of the media.

time format
Returns the current time format.

Comments

Before issuing any commands that use position values, you should set the desired time format using the **set** command.

Example

The following command returns the length of the waveform data in milliseconds (assuming the time format is set to milliseconds):

```
status mysound length
```

stop

Syntax

stop *device_id* [**notify**] [**wait**]

The **stop** command stops playing or recording.

Example

The following command stops playback or recording in the "mysound" device:

```
stop mysound
```

Chapter 8
Multimedia File Formats

This chapter describes the multimedia file formats. The chapter describes the structure of each file type and includes detailed lists of the data structures and fields contained in the files. The chapter also presents examples of multimedia files.

Several multimedia file formats used with Windows are based on the Resource Interchange File Format (RIFF). This chapter defines RIFF, the preferred format for new multimedia file types. If your application requires a new file format, you should define it using the RIFF tagged file structure described in this chapter.

This chapter describes the following file formats:

- RIFF DIB File Format (RDIB)
- Musical Instrument Digital Interface (MIDI) File Format
- RIFF MIDI File Format (RMID)
- Palette File Format (PAL)
- Waveform Audio File Format (WAVE)

For example, the four-character code "SMP" is stored as a sequence of four bytes ('S' 'M' 'P' ' ') in ascending addresses. For quick comparisons, a four-character code can also be treated as a 32-bit number.

The chunk fields are as follows:

Part	Description
ckID	Chunk ID. This four-character code identifies the representation of the chunk data. A program reading a RIFF file can skip over any chunk whose chunk ID it doesn't recognize; it skips the number of bytes specified by the **ckSize** field plus the pad byte, if present.
ckSize	Chunk size. This is a 32-bit unsigned value identifying the size of **ckData**. This size value includes does not include the size of the **ckID** or **ckSize** fields or the pad byte at the end of the **ckData** field.
ckData	Chunk data. This is binary data of fixed or variable size. The start of **ckData** is word-aligned with the start of the RIFF file. If the size of the chunk is an odd number of bytes, a pad byte with value zero is written after **ckData**. Word aligning is done to improve access speed (for chunks resident in memory) and for compatibility with EA IFF. The **ckSize** value does not include the pad byte.

Two types of chunks, the "LIST" and "RIFF" chunks, may contain nested chunks, or subchunks. These special chunk types are discussed later in this document. All other chunk types store a single element of binary data in **ckData**.

Chunks are represented using the following notation (in this example, the **ckSize** field and pad byte are implicit):

```
<ckID> ( <ckData> )
```

For example, a chunk with chunk ID "SMP" might be represented as follows:

```
SMP (  <sample-Data> )
```

It's common to refer to chunks by their chunk ID; the chunk shown above would be called an "SMP" chunk.

Defining and Registering RIFF Forms

The form-type code for a RIFF form must be unique. To guarantee this uniqueness, you must register any new form types before release. To register form types, and to get a current list of registered RIFF forms, request a *Multimedia Developer Registration Kit* from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

Like RIFF forms, RIFX forms must also be registered. Registering a RIFF form does not automatically register the RIFX counterpart. No RIFX form types are currently defined.

When you document the RIFF form, you should use the notation described in "Notation for Representing RIFF Files," later in this chapter.

Registered Form and Chunk Types

By convention, the form-type code for registered form types contains only digits and uppercase letters. Form-type codes that are all uppercase denote a registered, unique form type. Use lowercase letters for temporary or prototype chunk types.

Certain chunk types are also globally unique and must also be registered before use. These registered chunk types are not specific to a certain form type; they can be used in any form. If a registered chunk type can be used to store your data, you should use the registered chunk type rather than define your own chunk type containing the same type of information.

For example, a chunk with chunk ID "INAM" always contains the name or title of a file. Also, within all RIFF files, filenames or titles are contained within chunks with ID "INAM" and have a standard data format.

Unregistered (Form-Specific) Chunk Types

Chunk types that are used only in a certain form type use a lowercase chunk ID. A lowercase chunk ID has specific meaning only within the context of a specific form type. After a form designer is allocated a registered form type, the designer can choose lowercase chunk types to use within that form. See the *Multimedia Developers Registration Kit* for details.

For example, a chunk with ID "scln" inside one form type might contain the "number of scan lines." Inside some other form type, a chunk with ID "scln" might mean "secondary lambda number."

Notation	Description

<number>[<modifier>]

A number consisting of an optional sign (+ or –) followed by one or more digits and modified by the optional **<modifier>**. Valid **<modifier>** values are as follows:

<modifier>	Meaning
none	16-bit number in decimal format
H	16-bit number in hexadecimal format
C	8-bit number in decimal format
CH	8-bit number in hexadecimal format
L	32-bit number in decimal format
LH	32-bit number in hexadecimal format

Several examples follow:

```
0
65535
-1
0L
4a3c89HL
-1C
21HC
```

Note that –1 and 65535 represent the same value. The application reading this file must know whether to interpret the number as signed or unsigned.

'<chars>'

A four-character code (32-bit quantity) consisting of a sequence of zero to four ASCII characters (**<chars>**) in the given order. If **<chars>** is less than four characters long, it is implicitly padded on the right with blanks. Two single quotes is equivalent to four blanks. Examples follow.

```
'RIFF'
'xyz'
''
```

<chars> can include escape sequences, which are combinations of characters introduced by a backslash (\) used to represent other characters. Escape sequences are listed in the following section.

Escape Sequences for Four-Character Codes and String Chunks

The following escape sequences can be used in four-character codes and string chunks:

Escape Sequence	ASCII Value (Decimal)	Description
\n	10	Newline character
\t	9	Horizontal tab character
\b	8	Backspace character
\r	13	Carriage return character
\f	12	Form feed character
\\	92	Backslash
\'	39	Single quotation mark
\"	34	Double quotation mark
ddd	Octal *ddd*	Arbitrary character

Extended Notation for Representing RIFF Form Definitions

When documenting RIFF forms that you create, use the notation listed in the preceding section along with the extended notation listed in the following table to unambiguously define the structure of the new form.

Notation	Description

<name>	A label that refers to some element of the file, where name is the name of the label, as in the following example:
	`<NAME-ck>`
	`<GOBL-form>`
	`<bitmap-bits>`
	`<smp>`
	Conventionally, a label that refers to a chunk is named <*ckID*-**ck**>, where *ckID* is the chunk ID. Similarly, a label that refers to a RIFF form is named <*formType*-**form**>, where *formType* is the name of the form's type.

Notation	Description
el1 | el2 | ... | elN	Exactly one of the listed elements must be present, as in the following example: `<hdr-ck> → hdr(<hdr-x> | <hdr-y> | <hdr-z>)` This example defines the "hdr" chunk's data as containing one of **<hdr-x>**, **<hdr-y>**, or **<hdr-z>**.
element...	One or more occurrences of **element** may be present. Note that an ellipsis has this meaning only if it follows an element; in other cases (such as, A | B | ... | Z), the ellipsis has its ordinary English meaning. If there is any possibility of confusion, an ellipsis should only be used to indicate one or more occurrences `<data-ck> → data(<count:INT> <item:INT>...)` This example defines the data of the "data" chunk to contain an integer **<count>**, followed by one or more occurrences of the integer **<item>**.
[element]...	Zero or more occurrences of element may be present, as in the following example: `<data-ck> → data(<count:INT> [<item:INT>]...)` This example defines the data of the "data" chunk to contain an integer **<count>** followed by zero or more occurrences of an integer **<item>**.

{ elements }	The group of **elements** within the braces should be considered a single element, as in the following example:
	`<blorg> → <this> | <that> | <other>...`
	This example defines **<blorg>** to be either **<this>** or **<that>** or one or more occurrences of **<other>**. The next example defines **<blorg>** to be either **<this>** or one or more occurrences of **<that>** or **<other>**, intermixed in any way.
	`<blorg> → <this> | {<that> | <other>}...`

Atomic Labels

The following are atomic labels, or labels that refer to primitive data types. Where available, the equivalent Microsoft C data type is also listed.

Label	Meaning	MS C Type
<CHAR>	8-bit signed integer	signed char
<BYTE>	8-bit unsigned quantity	unsigned char
<INT>	16-bit signed integer in Intel format	signed int
<WORD>	16-bit unsigned quantity in Intel format	unsigned int
<LONG>	32-bit signed integer in Intel format	signed long
<DWORD>	32-bit unsigned quantity in Intel format	unsigned long
<FLOAT>	32-bit IEEE floating point number	float
<DOUBLE>	64-bit IEEE floating point number	double
<STR>	String (a sequence of characters)	
<ZSTR>	NULL-terminated string	
<BSTR>	String with byte (8-bit) size prefix	
<WSTR>	String with word (16-bit) size prefix	
<BZSTR>	NULL-terminated string with byte size prefix	
<WZSTR>	NULL-terminated string with word size prefix	

NULL-terminated means that the string is followed by a character with ASCII value 0.

A size prefix is an unsigned integer, stored as a byte or a word in Intel format, that specifies the length of the string. In the case of strings with BZ or WZ modifiers, the size prefix specifies the size of the string without the terminating NULL.

Note The WINDOWS.H header file defines the C types BYTE, WORD, LONG, and DWORD. These types correspond to labels **<BYTE>**, **<WORD>**, **<LONG>**, and **<DWORD>**, respectively.

Since the definition of the GOBL form does not refer to the INFO chunk, software that expects only "org" and "obj" chunks in a GOBL form would ignore the unknown "INFO" chunk.

```
RIFF( 'GOBL'
        LIST('INFO'         // INFO list containing filename and copyright
                INAM("A House"Z)
                ICOP("(C) Copyright Joe Inc. 1991"Z)
            )

        org (2, 0, 0)       // origin of object list

        LIST('obj'          // object list containing two polygons
                poly(0,0,0  2,0,0  2,2,0, 1,3,0, 0,2,0)
                poly(0,0,5  2,0,5  2,2,5, 1,3,5, 0,2,5)
            )
)                           // end of form
```

Storing Strings in RIFF Chunks

This section lists methods for storing text strings in RIFF chunks. While these guidelines may not make sense for all applications, you should follow these conventions if you must make an arbitrary decision regarding string storage.

NULL-Terminated String (ZSTR) Format

A NULL-terminated string (ZSTR) consists of a series of characters followed by a terminating NULL character. The ZSTR is better than a simple character sequence (STR) because many programs are easier to write if strings are NULL-terminated. ZSTR is preferred to a string with a size prefix (BSTR or WSTR) because the size of the string is already available as the **<ckSize>** value, minus one for the terminating NULL character.

String Table Format

In a string table, all strings used in a structure are stored at the end of the structure in packed format. The structure includes fields that specify the offsets from the beginning of the string table to the individual strings, as in the following example:

```
typedef struct
{
    INT     iWidgetNumber;      // the widget number
    WORD    offszWidgetName;    // an offset to a string in <rgchStrTab>
    WORD    offszWidgetDesc;    // an offset to a string in <rgchStrTab>
    INT     iQuantity;          // how many widgets
    CHAR    rgchStrTab[1];      // string table (allocate as large as needed)
}   WIDGET;
```

LIST Chunk

A LIST chunk is defined as follows:

```
LIST( <list-type> [<chunk>]... )
```

A LIST chunk contains a list, or ordered sequence, of subchunks. The **<list-type>** is a four-character code that identifies the contents of the list.

If an application recognizes the list type, it should know how to interpret the sequence of subchunks. However, since a LIST chunk may contain only subchunks (after the list type), an application that does not know about a specific list type can still walk through the sequence of subchunks.

Like chunk IDs, list types must be registered, and an all-lowercase list type has meaning relative to the form that contains it.

The INFO List Chunk

The "INFO" list is a registered global form type that can store information that helps identify the contents of the chunk. This information is useful but does not affect the way a program interprets the file; examples are copyright information and comments. An "INFO" list is a "LIST" chunk with list type "INFO." The following example shows a sample "INFO" list chunk:

```
LIST('INFO' INAM("Two Trees"Z)
            ICMT("A picture for the opening screen"Z) )
```

An "INFO" list should contain only the following chunks. New chunks may be defined, but an application should ignore any chunk it doesn't understand. The chunks listed below may only appear in an "INFO" list. Each chunk contains a ZSTR, or null-terminated text string.

Chunk ID	Description
IARL	*Archival Location.* Indicates where the subject of the file is archived.
IART	*Artist.* Lists the artist of the original subject of the file; for example, "Michaelangelo."
ICMS	*Commissioned.* Lists the name of the person or organization that commissioned the subject of the file; for example, "Pope Julian II."
ICMT	*Comments.* Provides general comments about the file or the subject of the file. If the comment is several sentences long, end each sentence with a period. Do *not* include newline characters.

Chunk ID	Description
ISBJ	*Subject.* Describes the contents of the file, such as "Aerial view of Seattle."
ISFT	*Software.* Identifies the name of the software package used to create the file, such as "Microsoft WaveEdit."
ISHP	*Sharpness.* Identifies the changes in sharpness for the digitizer required to produce the file (the format depends on the hardware used).
ISRC	*Source.* Identifies the name of the person or organization who supplied the original subject of the file; for example, "Trey Research."
ISRF	*Source Form.* Identifies the original form of the material that was digitized, such as "slide," "paper," "map," and so on. This is not necessarily the same as IMED.
ITCH	*Technician.* Identifies the technician who digitized the subject file; for example, "Smith, John."

RIFF DIB File Format (RDIB)

The RDIB format consists of a Windows 3.0 or Presentation Manager 1.2 DIB enclosed in a "RIFF" chunk. Enclosing the DIB in a "RIFF" chunk allows the file to be consistently identified; for example, an "INFO" list can be included in the file.

The "RDIB" form is defined as follows, using the standard RIFF form definition notation:

```
<RDIB-form> →  RIFF (  'RDIB'              // RIFF header
                       data( <DIB-data> )) // bitmap data in DIB format
```

For information on the Windows DIB format, see the *Microsoft Windows Programmer's Reference, Volume 4: Resources.*

Fields for the **LOGPALETTE** structure are described in the following table:

Field	Description
palVersion	Specifies the Windows version number for the structure.
palNumEntries	Specifies the number of palette color entries.
palPalEntry[]	Specifies an array of PALETTEENTRY data structures that define the color and usage of each entry in the logical palette.

The colors in the palette entry table should appear in order of importance. This is because entries earlier in the logical palette are most likely to be placed in the system palette.

PALETTEENTRY Structure

The **PALETTEENTRY** data structure specifies the color and usage of an entry in a logical color palette. The structure is defined as follows:

```
typedef struct tagPALETTEENTRY {
    BYTE    peRed;
    BYTE    peGreen;
    BYTE    peBlue;
    BYTE    peFlags;
} PALETTEENTRY;
```

Fields for the **PALETTEENTRY** structure are defined in the following table:

Field	Description
peRed	Specifies the intensity of red for the palette entry color.
peGreen	Specifies the intensity of green for the palette entry color.
peBlue	Specifies the intensity of blue for the palette entry color.
peFlags	Specifies how the palette entry is to be used.

WAVE Chunk Descriptions

The chunks contained in the WAVE form definition are described in more detail in the following table:

Chunk ID	Description
fmt	This chunk contains **<wave-format>**, which specifies the format of the data contained in **<data-ck>**. The **<wave-format>** chunk contains a structure consisting of the following fields:

Field	Description
wFormatTag	A number which indicates the WAVE format category of the file. The content of the **<FormatSpecific>** field of the "fmt" chunk, and the interpretation of data in the "data" chunk, depend on this value. The valid values for **<wFormatTag>**, and a description of each WAVE format category, is given in the next section.
nChannels	The number of channels represented in **<data-ck>**, such as 1 for mono or 2 for stereo.
nSamplesPerSec	The sampling rate (in samples per second) that each channel should be played back at.
nAvgBytesPerSec	The average number of bytes per second that data in **<data-ck>** should be transferred at. If **<wFormatTag>** is WAVE_FORMAT_PCM, then **<nAvgBytesPerSec>** should be equal to the following formula: $\frac{nChannels \times nBitsPerSecond \times nBitsPerSample}{8}$ Playback software can estimate the buffer size using the **<nAvgBytesPerSec>** value.
nBlockAlign	The block alignment (in bytes) of the data in **<data-ck>**. If **<wFormatTag>** is set to WAVE_FORMAT_PCM, then **<nBlockAlign>** should be equal to the following formula: $\frac{nChannels \times nBitsPerSample}{8}$

The following illustrations show the data packing for a 8-bit mono and stereo WAVE files:

Sample 1	Sample 2	Sample 3	Sample 4
Channel 0	*Channel 0*	*Channel 0*	*Channel 0*

Data packing for 8-bit mono PCM.

Sample 1		Sample 2	
Channel 0 (left)	*Channel 1 (right)*	*Channel 0 (left)*	*Channel 1 (right)*

Data packing for 8-bit stereo PCM.

The following illustrations show the data packing for 16-bit mono and stereo WAVE files:

Sample 1		Sample 2	
Channel 0 low-order byte	*Channel 0 high-order byte*	*Channel 0 low-order byte*	*Channel 0 high-order byte*

Data packing for 16-bit mono PCM.

Sample 1			
Channel 0 (left) low-order byte	*Channel 0 (left) high-order byte*	*Channel 1 (right) low-order byte*	*Channel 1 (right) high-order byte*

Data packing for 16-bit stereo PCM.

Examples of WAVE Files

The following PCM WAVE file has a 11.025 kHz sampling rate, mono, 8 bits per sample:

```
RIFF('WAVE' fmt(1, 1, 11025, 11025, 1, 8)
            data( <wave-data> ) )
```

The following PCM WAVE file has a 22.05 kHz sampling rate, stereo, 8 bits per sample:

```
RIFF('WAVE' fmt(1, 2, 22050, 44100, 2, 8)
            data( <wave-data> ) )
```

The following PCM WAVE file has a 44.1 kHz sampling rate, mono, 20 bits per sample:

```
RIFF('WAVE' INFO(INAM("O Canada"Z))
            fmt(1, 1, 44100, 132300, 3, 20)
            data( <wave-data> ) )
```

Appendix A

MCI Command String Syntax Summary

This appendix provides a summary of the syntax of the MCI command strings. The following command tables are included in this chapter:

- System command set
- Required command set
- Basic command set
- Animation command set
- CD audio command set
- MIDI sequencer command set
- Videodisc command set
- Video overlay command set
- Waveform audio command set

For information on using MCI command strings and descriptions of the commands, see Chapter 7, "MCI Command Strings."

The following conventions are used in the previous example and for the other MCI command tables:

Type Style	Used For
Bold	A specific term intended to be used literally. When used in the command column, it represents the MCI string command. When used in the arguments column, it represents a flag. For example, the **capability** command and the **can play** flag must be typed as shown.
Italics	Placeholders for information you must provide. The MCI command or flag associated with the information must precede it. For example, to use the alias words for the *device_name* with the **capability** command, type "capability words can play."
|	Divider for mutually exclusive arguments. When multiple arguments are separated by this symbol, only one of them can be used for each command. For example, the items in the list **can eject**, **can play**, **can record**, **can save**, **compound device**, **device type**, **has audio**, **has video**, **uses files** are mutually exclusive and you must select only one to use with the **capability** command. (Do not type the | with the argument.)
{ }	Required argument. You must include an argument enclosed by braces. For example, you must use one of the arguments **can eject**, **can play**, **can record**, **can save**, **compound device**, **device type**, **has audio**, **has video**, **uses files** with the **capability** command. (Do not type the braces with the argument.)
[]	Optional argument. For example, the **alias, type** and **shareable** flags are optional for the **open** command. You can use any combination of these flags. You can also use optional arguments with a required argument. (Do not type the brackets with the argument.)

Command	Arguments
open *device_name*	[**alias** *device_alias*]
	[**shareable**]
	[**type** *device_type*]
status *device_name*	[**mode** | **ready**]

Basic Command Set

In addition to the commands described previously, each device supports a set of commands specific to its device type. Although these commands are optional for a device, if a command is used it must support the options listed in this table as a minimum set of capabilities. Basic commands also support **notify** and **wait** as optional flags. You can add either or both of these flags to any basic command.

Command	Arguments
load *device_name*	{*file_name*}
pause *device_name*	
play *device_name*	[**from** *position*]
	[**to** *position*]
record *device_name*	[**from** *position*]
	[**to** *position*]
	[**insert** | **overwrite**]
resume *device_name*	
save *device_name*	[*file_name*]
seek *device_name*	{**to** *position* | **to start** | **to end**}

Animation Command Set

Animation devices support the following set of commands. These devices also support **notify** and **wait** as optional flags. You can add either or both of these flags to any animation command.

Command	Arguments
capability *device_name*	**{can eject** **| can play** **| can record** **| can reverse** **| can save** **| can stretch** **| compound device** **| device type** **| fast play rate** **| has audio** **| has video** **| normal play rate** **| slow play rate** **| uses files** **| uses palettes** **| windows}**
close *device_name*	
info *device_name*	**[file | product | window text]**
open *device_name*	**[alias** *device_alias*] **[nostatic]** **[parent** *hwnd*] **[shareable]** **[style child** **| style overlapped** **| style popup]** **[type** *device_type*]
pause *device_name*	

Command	Arguments
status *device_name*	**{current track** **| forward** **| length** **| length track** *track_number* **| media present** **| mode** **| number of tracks** **| palette handle** **| position** **| position track** *track_number* **| ready** **| speed** **| start position** **| stretch** **| time format** **| window handle}**
step *device_name*	**[by** *frames*]
	[reverse]
stop *device_name*	
update *device_name*	**[at** *update_rect*]
	[hdc *hdc*]
where *device_name*	**{destination | source}**
window *device_name*	**[handle** *window_handle* **| handle default]**
	[stretch | fixed]
	[state hide **| state iconic** **| state maximized** **| state minimize** **| state minimized** **| state no action** **| state no activate** **| state normal** **| state show]**
	[text *caption_text*]

Command	Arguments
set *device_name*	**[audio all off** **| audio all on** **| audio left off** **| audio left on** **| audio right off** **| audio right on** **| video off** **| video on]** **[door closed | door open]** **[time format milliseconds** **| time format ms** **| time format msf** **| time format tmsf]**
status *device_name*	**{current track** **| length** **| length track** *track_number* **| media present** **| mode** **| number of tracks** **| position** **| position track** *track_number* **| ready** **| start position** **| time format}**
stop *device_name*	

Command	Arguments
set *device_name*	**[audio all off** **| audio all on** **| audio left off** **| audio left on** **| audio right off** **| audio right on** **| video off** **| video on]** **[master MIDI** **| master none** **| master SMPTE]** **[offset** *hmsf_value*] **[port** *port_number* **| port mapper** **| port none]** **[slave file** **| slave MIDI** **| slave none** **| slave SMPTE]** **[tempo** *tempo_value*] **[time format milliseconds** **| time format ms** **| time format smpte 24** **| time format smpte 25** **| time format smpte 30** **| time format smpte 30 drop** **| time format song pointer]**

Videodisc Command Set

Videodisc players support the following set of commands. Videodisc devices also support **notify** and **wait** as optional flags. You can add either or both of these flags to any videodisc command.

Command	Arguments
capability *device_name*	{**can eject** | **can play** | **can record** | **can reverse** | **can save** | **compound device** | **device type** | **fast play rate** | **has audio** | **has video** | **normal play rate** | **slow play rate** | **uses files**}
	[CAV | CLV]
close *device_name*	
escape *device_name*	{*command_string*}
info *device_name*	**[product]**
open *device_name*	**[alias** *device_alias*]
	[shareable]
pause *device_name*	
play *device_name*	**[fast | slow | speed** *fps*]
	[from *position*]
	[scan]
	[to *position* **| reverse]**
resume *device_name*	
seek *device_name*	**[reverse | to** *position* **| to start | to end]**

Video Overlay Command Set

Video overlay devices supports the following set of commands. Video overlay devices also support **notify** and **wait** as optional flags. You can add either or both of these flags to any video overlay command.

Command	Arguments
capability *device_name*	{**can eject** | **can freeze** | **can play** | **can record** | **can save** | **can stretch** | **compound device** | **device type** | **has audio** | **has video** | **uses files** | **windows**}
close *device_name*	
freeze *device_name*	[**at** *rectangle*]
info *device_name*	[**product** | **file** | **window text**]
load *device_name*	[*file_name*]
	at *buffer_rectangle*
open *device_name*	[**alias** *device_alias*]
	[**parent** *hwnd*]
	[**shareable**]
	[**style** *style_type* | **style child** | **style overlapped** | **style popup**]
	[**type** *device_type*]

Command	Arguments
window *device_name*	**[handle** *window_handle* **| handle default]** **[stretch | fixed]** **[state hide** **| state iconic** **| state maximized** **| state minimize** **| state minimized** **| state no action** **| state no activate** **| state normal** **| state show]** **[text** *caption_text*]

Waveform Audio Command Set

Wave audio devices support the following set of commands. Wave audio devices also support **notify** and **wait** as optional flags. You can add either or both of these flags to any wave audio command.

Command	Arguments
capability *device_name*	**{can eject** **| can play** **| can record** **| can save** **| compound device** **| device type** **| has audio** **| has video** **| inputs** **| outputs** **| uses files}**
close *device_name*	
cue *device_name*	**[input | output]**

Command	Arguments
set *device_name*	**[alignment** *block_alignment*]
	[any input]
	[any output]
	[audio all off **| audio all on** **| audio left off** **| audio left on** **| audio right off** **| audio right on** **| video off** **| video on]**
	[bitspersample *bit_count*]
	[bytespersec *byte_rate*]
	[channels *channel_count*]
	[format tag *tag* **| format tag pcm]**
	[input *device_number*]
	[output *device_number*]
	[samplespersec *sample_rate*]
	[time format milliseconds **| time format ms** **| time format bytes** **| time format samples]**

Appendix B

Manufacturer ID and Product ID Lists

This appendix provides lists of the manufacturer and product IDs currently used with the multimedia extensions of Windows. This list will grow as more manufacturers create multimedia products for Windows.

To get a current list of manufacturer and product IDs for the multimedia Extensions, and to register new ones, request a *Multimedia Developer Registration Kit* from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

Index

J

L

M

N

O

P

T

U

V

W