Multimedia Programming Interface and Data Specifications 1.0

Issued as a joint design by IBM Corporation and Microsoft Corporation

August 1991

This document describes the programming interfaces and data specifications for multimedia that are common to both OS/2 and Windows environments. These specifications may be enhanced to incorporate new technologies or modified based on customer feedback and, as such, specifications incorporated into any final product may vary.

Contents

Chapter 1

Overview of Multimedia Specifications

This document describes the file format and control interface specifications for multimedia. These specifications allow developers to use common file format and device control interfaces.

Resource Interchange File Format

The Resource Interchange File Format (RIFF), a tagged file structure, is a general specification upon which many file formats can be defined. The main advantage of RIFF is its extensibility; file formats based on RIFF can be future-proofed, as format changes can be ignored by existing applications.

The RIFF file format is suitable for the following multimedia tasks:

- Playing back multimedia data
- Recording multimedia data
- Exchanging multimedia data between applications and across platforms

Chapter 2, "Resource Interchange File Format," describes the RIFF format.

Multimedia File Formats

A number of RIFF-based and non-RIFF file formats have been defined for the storage of multimedia data. Chapter 3, "Multimedia File Formats," describes the following file formats:

- Bundle File Format
- Device-Independent Bitmap (DIB) and RIFF DIB file formats
- Musical Instrument Digital Interface (MIDI) and RIFF MIDI file formats
- Palette File Format
- Rich Text File Format
- Waveform Audio File Format

Media Control Interface

The Media Control Interface (MCI) is a high-level control mechanism that provides a device-independent interface to multimedia devices and resource files.

The Media Control Interface (MCI) provides a command set for playing and recording multimedia devices and resource files. Developers creating multimedia applications are encouraged to use this high-level command interface rather than the low-level functions specific to each platform. The MCI command set acts as a platform-independent layer that sits between multimedia applications and the underlying system software.

The MCI command set is extensible in two ways:

- Developers can incorporate new multimedia devices and file formats in the MCI command set by creating new MCI drivers to interpret the commands.
- New commands and command options can be added to support special features or functions required by new multimedia devices or file formats.

Using MCI, an application can control multimedia devices using simple command strings like open, play, and close. The MCI command strings provide a generic interface to different multimedia devices, reducing the number of commands a developer needs to learn. A multimedia application might even accept MCI commands from an end user and pass them unchanged to the MCI driver, which parses the command and performs the appropriate action.

Chapter 3, "Media Control Interface," describes MCI and its command set in detail.

Registering Multimedia Formats

This document discusses several multimedia codes and formats that require registration. These multimedia elements include the following:

- Compression techniques
- RIFF form types, chunk IDs, and list types
- Compound-file usage codes
- Waveform audio format codes

To register these multimedia elements, request a *Multimedia Developer Registration Kit* from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

The *Multimedia Developer Registration Kit* also lists currently defined multimedia elements.

Chapter 2

Resource Interchange File Format

The Resource Interchange File Format (RIFF) is a tagged file structure developed for use on multimedia platforms. This chapter defines RIFF and describes the file structures based on RIFF. If your application requires a new file format, you should define it using the RIFF tagged file structure described in this chapter.

About the RIFF Tagged File Format

RIFF (Resource Interchange File Format) is the tagged file structure developed for multimedia resource files. The structure of a RIFF file is similar to the structure of an Electronic Arts IFF file. RIFF is not actually a file format itself (since it does not represent a specific kind of information), but its name contains the words "interchange file format" in recognition of its roots in IFF. Refer to the EA IFF definition document, *EA IFF 85 Standard for Interchange Format Files*, for a list of reasons to use a tagged file format.

RIFF has a counterpart, RIFX, that is used to define RIFF file formats that use the Motorola integer byte-ordering format rather than the Intel format. A RIFX file is the same as a RIFF file, except that the first four bytes are 'RIFX' instead of 'RIFF', and integer byte ordering is represented in Motorola format.

Notation Conventions

The following table lists some of the notation conventions used in this document. Further conventions and the notation for documenting RIFF forms are presented later in the document in the section "Notation for Representing Sample RIFF Files."

Notation	Description
<element label>	RIFF file element with the label "element label"
<element label: TYPE>	RIFF file element with data type "TYPE"
[<**element label**>]	Optional RIFF file element
<**element label**>...	One or more copies of the specified element
[<**element label**>]...	Zero or more copies of the specified element

Chunks

The basic building block of a RIFF file is called a *chunk*. Using C syntax, a chunk can be defined as follows:

```
typedef unsigned long DWORD;
typedef unsigned char BYTE;

typedef DWORD FOURCC;             // Four-character code

typedef FOURCC CKID;              // Four-character-code chunk identifier
typedef DWORD CKSIZE;             // 32-bit unsigned size value

typedef struct {                  // Chunk structure
      CKID       ckID;                  // Chunk type identifier
      CKSIZE     ckSize;                // Chunk size field (size of ckData)
      BYTE       ckData[ckSize];        // Chunk data
} CK;
```

A FOURCC is represented as a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters (ASCII character value 32) as required, with no embedded blanks.

For example, the four-character code ‘FOO’ is stored as a sequence of four bytes: 'F', 'O', 'O', ' ' in ascending addresses. For quick comparisons, a four-character code may also be treated as a 32-bit number.

The three parts of the chunk are described in the following table:

Part	Description
ckID	A four-character code that identifies the representation of the chunk data **data**. A program reading a RIFF file can skip over any chunk whose chunk ID it doesn't recognize; it simply skips the number of bytes specified by **ckSize** plus the pad byte, if present.
ckSize	A 32-bit unsigned value identifying the size of **ckData**. This size value does not include the size of the **ckID** or **ckSize** fields or the pad byte at the end of **ckData**.
ckData	Binary data of fixed or variable size. The start of **ckData** is word-aligned with respect to the start of the RIFF file. If the chunk size is an odd number of bytes, a pad byte with value zero is written after **ckData**. Word aligning improves access speed (for chunks resident in memory) and maintains compatibility with EA IFF. The **ckSize** value does not include the pad byte.

We can represent a chunk with the following notation (in this example, the **ckSize** and pad byte are implicit):

```
<ckID> ( <ckData> )
```

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**>.

RIFF Forms

A RIFF form is a chunk with a 'RIFF' chunk ID. The term also refers to a file format that follows the RIFF framework. The following is the current list of registered RIFF forms. Each is described in Chapter 3, "Multimedia File Formats."

Form Type	Description
PAL	RIFF Palette Format
RDIB	RIFF Device Independent Bitmap Format
RMID	RIFF MIDI Format
RMMP	RIFF Multimedia Movie File Format
WAVE	Waveform Audio Format

Using the notation for representing a chunk, a RIFF form looks like the following:

```
RIFF ( <formType> <ck>... )
```

The first four bytes of a RIFF form make up a chunk ID with values 'R', 'I', 'F', 'F'. The **ckSize** field is required, but for simplicity it is omitted from the notation.

The first DWORD of chunk data in the 'RIFF' chunk (shown above as **<formType>**) is a four-character code value identifying the data representation, or *form type*, of the file. Following the form-type code is a series of subchunks. Which subchunks are present depends on the form type. The definition of a particular RIFF form typically includes the following:

- A unique four-character code identifying the form type
- A list of mandatory chunks
- A list of optional chunks
- Possibly, a required order for the chunks

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. See "Registering Multimedia Formats" in Chapter 1, "Overview of Multimedia Specifications," for information on registering RIFF forms.

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.

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 "Registering Multimedia Formats" in Chapter 1, "Overview of Multimedia Specifications," for information on registering form types.

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 for Representing Sample RIFF Files

RIFF is a binary format, but it is easier to comprehend an ASCII representation of a RIFF file. This section defines a standard notation used to present samples of various types of RIFF files. If you define a RIFF form, we urge you to use this notation in any file format samples you provide in your documentation.

Basic Notation for Representing RIFF Files

The following table summarizes the elements of the RIFF notation required for representing sample RIFF files:

Notation	Description

<ckID> (<ckData>)	The chunk with ID **<ckID>** and data **<ckData>**. As previously described, **<ckID>** is a four-character code which may be enclosed by single quotes for emphasis.
	For example, the following notation describes a 'RIFF' chunk with a form type of 'QRST'. The data portion of this chunk contains a 'FOO' subchunk.
	`RIFF('QRST' FOO(17 23))`
	The following example describes an 'ICOP' chunk containing the string " Copyright Encyclopedia International." :
	`'ICOP' ("Copyright Encyclopedia International."Z)`

<number>[<modifier>]	A number in Intel format, where **<number>** is an optional sign (+ or -) followed by one or more digits and modified by the optional **<modifier>**. Valid **<modifier>** values follow:

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 (\) and used to represent other characters. Escape sequences are listed in the following section.

"<string>"[<modifier>] The sequence of ASCII characters contained in **<string>** and modified by the optional modifier **<modifier>**. The quoted text can include any of the escape sequences listed in the following section. Valid **<modifier>** values follow:

Modifier	Meaning
none	No NULL terminator or size prefix.
Z	String is NULL-terminated
B	String has an 8-bit (byte) size prefix
W	String has a 16-bit (word) size prefix
BZ	String has a byte-size prefix and is NULL-terminated
WZ	String has a word-size prefix and is NULL-terminated

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 preceding the string characters, 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.

The various string formats referred to above are discussed in "Storing Strings in RIFF Chunks," following later in this section., +

Examples follow:

```
"No prefix, no NULL terminator"
"No prefix, NULL terminator"Z
"Byte prefix, NULL terminator"BZ
```

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	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 quote
\"	34	Double quote
ddd	Octal *ddd*	Arbitrary character

Extended Notation for Representing RIFF Form Definitions

To unambiguously define the structure of new RIFF forms, document the RIFF form using the basic notation along with the following extended notation:

Notation	Description

<name>	A label that refers to some element of the file, where **<name>** is the name of the label. Examples follow:
	`<NAME-ck>`
	`<GOBL-form>`
	`<bitmap-bits>`
	`<foo>`
	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.

<name> → elements	The actual data represented by **<name>** is defined as **elements**. This states that **<name>** is an abbreviation for **elements**, where **elements** is a sequence of other labels and literal data. An example follows: `<GOBL-form> → RIFF ( 'GOBL' <form-data> )` This example defines label <GOBL-form> as representing a RIFF form with chunk ID 'GOBL' and data equal to <form-data>, where <form-data> is a label that would be defined in another rule. Note that a label may represent any data, not just a RIFF chunk or form. ***Note:*** A number of atomic labels are defined in the section " Atomic Labels" later in this document. These labels refer to primitive data types.

<name:type>	This is the same as **<name>**, but it also defines **<name>** to be equivalent to **<type>**. This notation obviates the following rule:
	`<name> → <type>`
	This allows you to give a symbolic name to an element of a file format and to specify the element data type. An example follows:
	`<xyz-coordinate> → <x:INT> <y:INT> <z:INT>`
	This defines <xyz-coordinate> to consist of three parts concatenated together: <x>, <y>, and <z>. The definition also specifies that <x>, <y>, and <z> are integers. This notation is equivalent to the following:
	`<xyz-coordinate> → <x> <y> <z>`
	`<x> → <INT>`
	`<y> → <INT>`
	`<z> → <INT>`

[elements]	

An optional sequence of labels and literal data. Surrounded by square brackets, it may be considered an element itself. An example follows:

```
<FOO-form> → RIFF('FOO' [<header-ck>] <data-ck>)
```

This example defines form "FOO" with an optional header chunk followed by a mandatory data chunk.

el1 | el2 | ... | elN

Exactly one of the listed elements must be present. An example follows:

```
<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. An ellipsis has this meaning only if it follows an element; in cases such as "el1 | el2 | ... | elN," 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. An example follows:

```
<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. An example follows.

```
<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. An example follows:

```
<blorg> → <this> | {<that> | <other>}...
```

This example defines <blorg> to be either <this> or one or more occurrences of <that> or <other>, intermixed in any way. Contrast this with the following example:

```
<blorg> → <this> | <that> | <other>...
```

This example defines <blorg> to be either <this> or <that> or one or more occurrences of <other>.

struct { ...} name

A structure defined using C syntax. This can be used instead of a sequence of labels if a C header (include) file is available that defines the structure. The label used to refer to the structure should be the same as the structure's typedef name. An example follows:

```
<3D_POINT> → struct {
                  INT x;              // x-coordinate
                  INT y;              // y-coordinate
                  INT z;              // z-coordinate
               } 3D_POINT
```

Wherever possible, the types used in the structure should be the types listed in the following section, " Atomic Labels," because these types are more portable than C types such as int. The structure fields are assumed to be present in the file in the order given, with no padding or forced alignment.

Unless the RIFF chunk ID is 'RIFX', integer byte ordering is assumed to be in Intel format.

// comment

An explanatory comment to a rule. An example follows:

```
<weekend> → 'Sat'|'Sun'       // Four-character code
                              // for day
```

Atomic Labels

The following are atomic labels, which are 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.

A Sample RIFF Form Definition and RIFF Form

The following example defines <GOBL-form>, the hypothetical RIFF form of type 'GOBL'. To fully document a new RIFF form definition, a developer would also provide detailed descriptions of each file element, including the semantics of each chunk and sample files documented using the standard notation.

```
<GOBL-form> →     RIFF( 'GOBL'               // RIFF form header
                        [<org-ck>]           // Origin chunk (default (0,0,0))
                        <obj-list>)          // Series of graphical objects

<org-ck> →        org(    <origin:3D_POINT> )        // Object-list origin

                                               // An object is a:
<obj-list> →      LIST(   'obj'   {   <sqr-ck>  |                 // square,
                                      <circ-ck> |                 // circle,
                                      <poly-ck>  }... )           // or polygon

<sqr-ck> →        sqr(    <pt1:3D_POINT>         // one vertex
                          <pt2:3D_POINT>         // another vertex
                          <pt3:3D_POINT> )       // a third vertex

<circ-ck> →       circ(   <center:3D_POINT>          // Center of circle
                          <circumPt:3D_POINT> )      // Point on circumference

<poly-ck> →       poly( <pt:3D_POINT>... )       // List of points in a polygon

<3D_POINT> →      struct                     // Defined in "gobl.h"
                  {   INT x;                     // x-coordinate
                      INT y;                     // y-coordinate
                      INT z;                     // z-coordinate
                  } 3D_POINT
```

Sample RIFF Form

The following sample RIFF form adheres to the form definition for form type GOBL. The file contains three subchunks:

- An 'INFO' list

- An ‘org’ chunk
- An ‘obj’ chunk

The ‘INFO’ list and ‘org’ chunk each have two subchunks. The ‘INFO’ list is a registered global chunk that can be used within any RIFF file. The ‘INFO’ list is described in the ‘INFO List Chunk,” later in this chapter.

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 Encyclopedia International 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. An example follows:

```
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;
```

If multiple chunks within the file need to reference variable-length strings, you can store the strings in a single chunk that acts as a string table. The chunks that refer to the strings contain offsets relative to the beginning of the data part of the string table chunk.

NULL-Terminated, Byte Size Prefix String (BZSTR) Series

In a BZSTR series, a series of strings is stored in packed format. Each string is a BZSTR, with a byte size prefix and a NULL terminator. This format retains the ease-of-use characteristics of the ZSTR while providing the string size, allowing the application to quickly skip unneeded strings.

Multiline String Format

When storing multiline strings, separate lines with a carriage return/line feed pair (ASCII 13/ASCII 10 pair). Although applications vary in their requirements for new line symbols (carriage return only, line feed only, or both), it is generally easier to strip out extra characters than to insert extra ones. Inserting characters might require reallocating memory blocks or pre-scanning the chunk before allocating memory for it.

Choosing a Storage Method

The following lists guidelines for deciding which storage method is appropriate for your application.

Usage	Recommended Format
Chunk data contains nothing except a string	ZSTR (NULL-terminated string) format.
Chunk data contains a number of fields, some of which are variable-length strings	String-table format
Multiple chunks within the file need to reference variable-length strings	String-table format
Chunk data stores a sequence of strings, some of which the application may want to skip	BZSTR (NULL-terminated string with byte size prefix) series

Chunk data contains multiline strings	A multiline string format

LIST Chunk

A LIST chunk contains a list, or ordered sequence, of subchunks. A LIST chunk is defined as follows:

```
LIST( <list-type> [<chunk>]... )
```

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. See "Registering Multimedia Formats" in Chapter 1, "Overview of Multimedia Specifications," for information on registering list types.

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 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.
ICOP	*Copyright.* Records the copyright information for the file. For example, "Copyright Encyclopedia International 1991." If there are multiple copyrights, separate them by a semicolon followed by a space.
ICRD	*Creation date.* Specifies the date the subject of the file was created. List dates in year-month-day format, padding one-digit months and days with a zero on

	the left. For example, "1553-05-03" for May 3, 1553.
ICRP	*Cropped.* Describes whether an image has been cropped and, if so, how it was cropped. For example, "lower right corner."
IDIM	*Dimensions.* Specifies the size of the original subject of the file. For example, "8.5 in h, 11 in w."
IDPI	*Dots Per Inch.* Stores dots per inch setting of the digitizer used to produce the file, such as "300."
IENG	*Engineer.* Stores the name of the engineer who worked on the file. If there are multiple engineers, separate the names by a semicolon and a blank. For example, "Smith, John; Adams, Joe."
IGNR	*Genre.* Describes the original work, such as, "landscape," "portrait," "still life," etc.
IKEY	*Keywords.* Provides a list of keywords that refer to the file or subject of the file. Separate multiple keywords with a semicolon and a blank. For example, "Seattle; aerial view; scenery."
ILGT	*Lightness.* Describes the changes in lightness settings on the digitizer required to produce the file. Note that the format of this information depends on hardware used.
IMED	*Medium.* Describes the original subject of the file, such as, "computer image," "drawing," "lithograph," and so forth.
INAM	*Name.* Stores the title of the subject of the file, such as, "Seattle From Above."
IPLT	*Palette Setting.* Specifies the number of colors requested when digitizing an image, such as "256."
IPRD	*Product.* Specifies the name of the title the file was originally intended for, such as "Encyclopedia of Pacific Northwest Geography."
ISBJ	*Subject.* Describes the conbittents 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 forth. This is not necessarily the same as IMED.
ITCH	*Technician.* Identifies the technician who digitized the subject file. For example, "Smith, John."

CSET (Character Set) Chunk

To define character-set and language information for a RIFF file, use the CSET chunk. The CSET chunk defines the code page and country, language, and dialect codes for the file. These values can be overridden for specific file elements; see "Usage Codes for Extra Header and Extra Entry Fields," later in this chapter, for information on specifying character set information in a compound file.

The CSET chunk is defined as follows:

```
<CSET chunk> → CSET( <wCodePage:WORD>
                     <wCountryCode:WORD>
                     <wLanguageCode:WORD>
                     <wDialect:WORD> )
```

The fields are as follows:

Field	Description
wCodePage	Specifies the code page used for file elements. If the CSET chunk is not present, or if this field has value zero, assume standard ISO 8859/1 code page (identical to code page 1004 without code points defined in hex columns 0, 1, 8, and 9).
wCountryCode	Specifies the country code used for file elements. See "Country Codes," following this section, for a list of currently defined country codes. If the CSET chunk is not present, or if this field has value zero, assume USA (country code 001).
wLanguage, **wDialect**	Specify the language and dialect used for file elements. See "Language and Dialect Codes," later in this chapter, for a list of language and dialect codes. If the CSET chunk is not present, or if these fields have value zero, assume US English (language code 9, dialect code 1).

Country Codes

Use one of the following country codes in the **wCountryCode** field:

Country Code	Country
000	None (ignore this field)
001	USA
002	Canada
003	Latin America
030	Greece
031	Netherlands

032	Belgium
033	France
034	Spain
039	Italy
041	Switzerland
043	Austria
044	United Kingdom
045	Denmark
046	Sweden
047	Norway
049	West Germany
052	Mexico
055	Brazil
061	Australia
064	New Zealand
081	Japan
082	Korea
086	People's Republic of China
088	Taiwan
090	Turkey
351	Portugal
352	Luxembourg
354	Iceland
358	Finland

Language and Dialect Codes

Specify one of the following pairs of language-code and dialect-code values in the **wLanguage** and **wDialect** fields:

Language Code	Dialect Code	Language
0	0	None (ignore these fields)
1	1	Arabic
2	1	Bulgarian
3	1	Catalan
4	1	Traditional Chinese
4	2	Simplified Chinese
5	1	Czech
6	1	Danish
7	1	German
7	2	Swiss German
8	1	Greek
9	1	US English
9	2	UK English
10	1	Spanish
10	2	Spanish Mexican
11	1	Finnish
12	1	French
12	2	Belgian French
12	3	Canadian French
12	4	Swiss French
13	1	Hebrew

14	1	Hungarian
15	1	Icelandic
16	1	Italian
16	2	Swiss Italian
17	1	Japanese
18	1	Korean
19	1	Dutch
19	2	Belgian Dutch
20	1	Norwegian - Bokmal
20	2	Norwegian - Nynorsk
21	1	Polish
22	1	Brazilian Portuguese
22	2	Portuguese
23	1	Rhaeto-Romanic
24	1	Romanian
25	1	Russian
26	1	Serbo-Croatian (Latin)
26	2	Serbo-Croatian (Cyrillic)
27	1	Slovak
28	1	Albanian
29	1	Swedish
30	1	Thai
31	1	Turkish
32	1	Urdu
33	1	Bahasa

JUNK (Filler) Chunk

A JUNK chunk represents padding, filler or outdated information. It contains no relevant data; it is a space filler of arbitrary size. The JUNK chunk is defined as follows:

```
<JUNK chunk> → JUNK( <filler> )
```

where **<filler>** contains random data.

Compound File Structure

The compound file structure is a RIFF-based structure upon which multimedia file formats can be defined. The compound file structure is a parameterized structure that provides for the following:

- Storage of multimedia data elements
- Direct access to multimedia data elements (as opposed to sequential searching)

The goals of the compound file structure are to maximize flexibility and extensibility while minimizing implementation costs. Using the compound file structure, developers of multimedia data formats can define both simple and complex file formats.

The structure is flexible enough to be used for many purposes, but it can be simplified for use with simple file formats. Designers of new multimedia file formats can restrict the use of standard header fields, requiring some and removing others.

For example, a developer might define a compound file format that stores a series of bitmaps in a single file, thus reducing compact disc seek times. Another developer might define a compound file format that contains a special type of audio resource, using the compound file header information to identify the attributes of the audio data stored within.

Structural Overview

Files based upon the compound file structure contain the following two RIFF chunks at their topmost level:

- Compound File Table of Contents (CTOC) chunk
- Compound File Element Group (CGRP) chunk

The CTOC chunk indexes the CGRP chunk, which contains the actual multimedia data elements. Defined using the standard chunk notation, a compound file is represented as follows:

```
<compound file> → RIFF('type' <CTOC> <CGRP>)
```

where 'type' is a FOURCC indicating the file type.

This section describes the CTOC and CGRP chunks in detail.

Compound File Table of Contents (CTOC) Chunk

The CTOC chunk functions mainly as an index, allowing direct access to elements within a compound file. The CTOC chunk also contains information about the attributes of the entire file and of each media element within the file.

To provide the maximum flexibility for defining compound file formats, the CTOC chunk can be customized at several levels. The CTOC chunk contains fields whose length and usage is defined by other CTOC fields. This parameterization adds complexity, but it provides flexibility to file format designers and allows applications to correctly read data without necessarily knowing the specific file format definition.

Structural Overview

The CTOC chunk defines the contents of the CGRP chunk. The CTOC chunk has the following components:

- Header information defining the size of the CTOC chunk, the number of entries in the CGRP chunk, the size of the CGRP chunk, and general information about the entire header file
- A parameter table definition defining the size and contents of the header parameter table and CTOC table entries

- A header parameter table defining attributes that apply to the entire compound file.
- CTOC table entries defining the location, size, name, and attributes of the compound file elements contained in the CGRP chunk.

These components appear sequentially in the CTOC chunk. The individual fields in the CTOC chunk are the following:

```
<CTOC-chunk>→ CTOC (
      <dwHeaderSize:DWORD>               // Header information
      <dwEntriesTotal:DWORD>
      <dwEntriesDeleted:DWORD>
      <dwEntriesUnused:DWORD>
      <dwBytesTotal:DWORD>
      <dwBytesDeleted:DWORD>
      <dwHeaderFlags:DWORD>

      <wEntrySize:WORD>                  // Parameter table definition
      <wNameSize:WORD>
      <wExHdrFields:WORD>
      <wExEntFields:WORD>
      <awExHdrFldUsage:WORD[wExHdrFields]>
      <awExEntFldUsage:WORD[wExEntFields]>

                                         // Header parameter table
      <adwExHdrField:DWORD[wExHdrFields]>
      [<bHeaderPad:BYTE>]
      [<CTOC-table-entry>]               // CTOC table entries
)
```

Each CTOC table entry is defined as follows:

```
<CTOC-table-entry> →
      <dwOffset:DWORD>
      <dwSize:DWORD>
      <dwMedType:DWORD>
      <dwMedUsage:DWORD>
      <dwCompressTech:DWORD>
      <dwUncompressBytes:DWORD>
      <adwExEntField:DWORD[wExEntFields]>
      <bEntryFlags:BYTE>
      <achName:CHAR[wNameSize]>
      [<bEntryPad:BYTE>]...
```

The following sections describe each field in detail.

Header Information

The header information section defines general information about the CTOC header and about the entire compound file. It contains the following fields:

Field Name	Description
dwHeaderSize	Combined size of header information, parameter table definition, and header parameter table. Use this value to locate the start of the CTOC table entries within the CTOC chunk.
dwEntriesTotal	Total number of CTOC table entries, including unused entries and entries corresponding to deleted elements.
dwEntriesDeleted	Number of CTOC table entries that correspond to deleted elements.
dwEntriesUnused	Number of CTOC table entries that are unused.
dwBytesTotal	Combined size of all CGRP elements, including deleted elements.
dwBytesDeleted	Combined size of all deleted CGRP elements.
dwHeaderFlags	Flags that give information about the entire compound file. The following flags may be used: CTOC_HF_SEQUENTIAL Valid CTOC table entries are arranged in sequential order. If this flag is not set, the CTOC table entries may be in an arbitrary order. CTOC_HF_MEDSUBTYPE The **dwMedUsage** field of each CTOC table entry contains a FOURCC that indicates how the element is used. If this flag is not set, the **dwMedUsage** field contains information as defined by the form type.

Parameter Table Definition

The parameter table definition defines the size and contents of the header parameter table and CTOC table. It contains the following fields:

Field Name	Description
wEntrySize	Size of each CTOC table entry, including any pad bytes.
wNameSize	Size of the **achName** field of each CTOC table entry. Each **achName** field must be padded with null characters to this length. The **achName** field is a null-terminated string, so it always contains at least one trailing null character.
wExHdrFields	Number of extra header fields, or entries in the **awExHdrFldUsage** and **adwExHdrField** arrays.

wExEntFields	Number of extra entry fields, or entries in the **awExEntFldUsage** and **adwExHdrField** arrays.
awExHdrFldUsage	Array of extra header field usage fields. Each entry in this array corresponds to the same numbered entry in the **adwExHdrField** array and defines how that entry is interpreted. Valid usage codes for each field in this array are listed in "Usage Codes for Extra Header and Extra Entry Fields," later in this chapter. The number of WORDs in this array is defined by the **wExHdrFields** value.
awExEntFldUsage	Array of extra entry field usage fields. Each entry in this array corresponds to the same numbered entry in the **adwExEntField** array, present in each CTOC table entry, and defines how that entry is interpreted. Valid usage codes for each field in this array are listed in "Usage Codes for Extra Header and Extra Entry Fields," later in this chapter. The number of WORDs in this array is defined by the **wExEntFields** value.

Header Parameter Table

The header parameter table is an optional component generally used to define attributes of the entire compound file.

Field Name	Type
adwExHdrField	Extra header fields. The usage of each cell in the array is defined by the corresponding cell in the **awExHdrFldUsage** array. The number of DWORDs in this array is defined by the **wExHdrFields** value.
bHeaderPad	Zero or more NULL pad bytes. There must be enough padding in this field to make the CTOC header an even number of bytes in length.

CTOC Table Entries

The CTOC table entries define the location, size, name, and other information about the individual compound file elements contained in the CGRP chunk. The number of CTOC table entries is determined by the **dwEntriesTotal** field in the header information of the CTOC chunk.

Each CTOC table entry is a structure containing the following fields:

Field Name	Description
dwOffset	Byte offset of the compound file element measured from the beginning of the data portion of the CGRP chunk. For example, if **dwOffset** is 1000 and the chunk ID of the CGRP chunk is at offset 500, the element is at offset 1508 (1000+500+4 (chunk ID)+4 (chunk size field)).
dwSize	Size of the element in bytes.
dwMedType	FOURCC value identifying the media element type of the compound file element. This field may be zero if the compound file element is not to be interpreted as a standalone file. If the compound file element is a RIFF form, then the media element type is the same as the RIFF form type.
dwMedUsage	Extra usage information for the compound file element. If the CTOC_HF_MEDSUBTYPE flag is set in the **dwHeaderFlags** field, this field contains a FOURCC that indicates how the element is used. To avoid name conflicts, this FOURCC must be registered. See "Registering Multimedia Formats" in Chapter 1, "Overview of Multimedia Specifications," for information on usage codes. If the CTOC_HF_MEDSUBTYPE flag is not set in the **dwHeaderFlags** field, this field contains 32 bits of information interpreted as defined by the form type.
dwCompressTech	Compression technique used to compress the media element. If this value is zero, the element is not compressed. See "Compression of Compound File Elements," later in this chapter, for more information.
dwUncompressBytes	Number of bytes the compound file element occupies in memory after decompression. This value assumes the decompression technique identified in the **dwCompressTech** field. If the **dwCompressTech** field is 0, then the compound file element is not compressed, and this field should equal the **dwSize** field.
adwExEntField	Array of extra entry fields defining attributes of this compound file element. The usage of each cell in the array is defined by the corresponding cell in the **awExEntFldUsage** array. The number of DWORDs in this array is defined by the **wExEntFields** value.
bEntryFlags	Flags giving information about the compound file element or this CTOC table entry. Possible values follow; these may be combined: CTOC_EF_DELETED Compound file is marked as deleted and should not be accessed. Do not combine this flag with the CTOC_EF_UNUSED flag. CTOC_EF_UNUSED CTOC table entry is unused and does not refer to any compound file element. This entry can be used to refer to a new compound file element. Do not combine this flag with the CTOC_EF_DELETED flag.
achName	Array of characters containing the name of the compound file

	element. The number of bytes in this array is defined by the **wNameSize** value. The string must be padded on the right with NULL characters and must be terminated by at least one NULL character. This field must be an odd number of bytes in length and must be at least one byte long.
bEntryPad	Zero or more NULL pad bytes as needed to make the table entry an even number of bytes in length.

Usage Codes for Extra Header and Extra Entry Fields

The following are valid usage codes for elements in the **awExHdrFldUsage** and **awExEntFldUsage** arrays, both of which are fields of the CTOC header. These arrays define the meaning of data stored in the **adwExHdrField** and **adwExEntField** "extra fields." All usage codes apply to both header fields and entry fields, unless explicitly stated otherwise.

Values marked in the extra header field arrays generally apply to all elements in the CFRG chunk, while values marked in the extra entry field arrays generally apply only to the element referenced by the corresponding CTOC table entry.

Flag	Description
CTOC_EFU_UNUSED (0x00)	The field is unused. This usage code may be used to logically delete a header field.
CTOC_EFU_LASTMODTIME (0x01)	When used to describe an extra header field, the field contains the time that any portion of the CTOC or CGRP was last modified. When used to describe an extra entry field, the field contains the time that the corresponding CTOC table entry, or the compound file element it refers to, was last modified. The field is interpreted as a DWORD containing the number of seconds that have elapsed since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970.
CTOC_EFU_CODEPAGE	The field contains the code page and country code for the **achName** field. These values override any values specified in a CSET chunk. When used to describe an extra header field, the field contains code-page and country-code information for all CTOC table entries. When used to describe an extra entry field, the field contains information for that specific CTOC table entry. The low-order word of the field contains one of the following code page values: Zero Use standard ISO 8859/1 code page. This is identical to code page 1004 without code points defined in hex columns 0, 1, 8, and 9. CTOC_CHARSET_CODEPAGE (0x0000*nnnn*)

Use code page 0x*nnnn*, where 0x*nnnn* is the 16-bit code page number. For example, 0x00000352 for OS/2 code page 850, or 0x000004E4 for Windows 3.1 code page 1252.

The high-order word contains one of the following country codes:

Zero
Ignore this field.

Country code
See "Country Codes," earlier in this chapter, for a list of currently defined country codes.

CTOC_EFU_LANGUAGE

The field contains language and dialect information for the **achName** field. These values override any values specified in a CSET chunk.

When used to describe an extra header field, the field contains language information for all CTOC table entries. When used to describe an extra entry field, the field contains information for that specific CTOC table entry.

The low-order word of the field contains one of the following language codes:

Zero
Ignore this field.

Language code
See "Language and Dialect Codes," earlier in this chapter, for a list of currently defined language codes.

The high-order word of the field contains one of the following dialect codes:

Zero
Ignore this field.

Dialect code
See "Language and Dialect Codes," earlier in this chapter, for a list of currently defined dialect codes.

CTOC_EFU_COMPRESSPARAM0 (0x05) through CTOC_EFU_COMPRESSPARAM9 (0x14)

Specifies a compression parameter. See "Compression of Compound File Elements," later in this chapter.

Compression of Compound File Elements

Compound file elements can be compressed. The **dwCompressTech** field of a CTOC table entry contains a FOURCC compression technique identifier for the corresponding compound file element. If the field is zero, the compound file element is not compressed.

The definition of a specific compression technique may specify that either the entire compound file element is compressed, or that some specific subset, for example one or more RIFF chunks, is compressed.

The **dwUncompressSize** field contains the number of bytes that the compound file element will occupy in memory after decompression. If the compound file element is not compressed, this field contain the same value as the **dwSize** field, which identifies the file size of the compound file element.

Compression techniques may demand extra header fields or extra entry fields for decompression parameters. Compression technique identifiers, and any new entry fields corresponding to decompression technique parameters, must be unique. See "Registering Multimedia Formats" in Chapter 1, "Overview of Multimedia Specifications," for information on registering compression techniques.

Compound File Element Group (CGRP) Chunk

The actual elements of data referenced by the CTOC chunk are stored in a compound file Element Group (CGRP) chunk. The CGRP chunk contains all the compound file elements, concatenated together into one contiguous block of data. Some of the elements in the CGRP chunk might be unused, if the element was marked for deletion or was altered and stored elsewhere within the CGRP chunk.

Elements within the CGRP chunk are of arbitrary size and can appear in a specific or arbitrary order, depending upon the file format definition. Each element is identified by a corresponding CTOC table entry.

Using the standard RIFF notation, the CGRP chunk is defined as follows:

```
<CGRP-chunk> → CGRP([<compound file element>]...)
```

Placement of the CTOC and CGRP Chunks

The specific file format definition can specify which of the two chunks appear first the data file. Generally, the CTOC chunk is placed at the front of the file to reduce the seek and read times required to access it. During authoring time, an application might place the CTOC chunk at the end of the file, so it can be expanded as elements are added to the CGRP chunk.

Chapter 3

Multimedia File Formats

This chapter describes the multimedia file formats. Most of these file formats are based on the Resource Interchange File Format (RIFF), described in Chapter 2.

This chapter describes the following file formats:

- Bundle File Format (BND)
- Device Independent Bitmap File Format (DIB)
- RIFF DIB File Format (RDIB)
- Musical Instrument Digital Interface File Format (MIDI)
- RIFF MIDI File Format (RMID)
- Palette File Format (PAL)
- Rich Text Format (RTF)
- Waveform Audio File Format (WAVE)

Bundle File Format

The Bundle (BND) format contains a series of RIFF chunks or other multimedia files. The BND file is defined as follows:

```
<BND-file> → RIFF('BND' <CTOC-chunk> <CGRP-chunk> )
```

The **<CTOC-chunk>** and **<CGRP-chunk>** formats are defined in "Compound File Structure," in Chapter 2, "Resource Interchange File Format."

Each compound file element must be capable of standing alone as an independent file. An element may not be a random chunk (except the RIFF chunk, indicating a RIFF file) or random binary data (unless the binary data is supposed to be treated as a file).

Device Independent Bitmap File Format

The Device Independent Bitmap (DIB) format represents bitmap images in a device-independent manner. Bitmaps can be represented at 1, 4, and 8 bits per pixel, with a palette containing colors represented in 24 bits. Bitmaps can also be represented at 24 bits per pixel without a palette and in a run-length encoded format.

This documentation describes three types of DIB files:

- Windows version 3.0 device-independent bitmap files
- OS/2 Presentation Manager version 1.2 device-independent bitmap files
- RIFF device-independent bitmap files

The Windows 3.0 and Presentation Manager 1.2 DIBs are similar, so they are discussed together.

Overview of DIB Structure

Windows 3.0 and Presentation Manager 1.2 DIB files consist of the following sequence of data structures:

- A file header
- A bitmap information header
- A color table
- An array of bytes that defines the bitmap bits

The following sections describe each of these structures.

Bitmap File Header

The bitmap file header contains information about the type, size, and layout of a device-independent bitmap (DIB) file. In both the Windows 3.0 and Presentation Manager 1.2 DIBs, it is defined as a BITMAPFILEHEADER data structure:

```
typedef struct tagBITMAPFILEHEADER {
      WORD    bfType;
      DWORD   bfSize;
      WORD    bfReserved1;
      WORD    bfReserved2;
      DWORD   bfOffBits;
} BITMAPFILEHEADER;
```

The following table describes the fields.

Field	Description
bfType	Specifies the file type. It must consist of the character sequence BM

	(WORD value 0x4D42).
bfSize	Specifies the file size in bytes.
bfReserved1	Reserved. Must be set to zero.
bfReserved2	Reserved. Must be set to zero.
bfOffBits	Specifies the byte offset from the BITMAPFILEHEADER structure to the actual bitmap data in the file.

Bitmap Information Header

The BITMAPINFO and BITMAPCOREINFO data structures define the dimensions and color information for Windows 3.0 and Presentation Manager 1.2 DIBs, respectively. They are defined as follows:

Windows 3.0 DIB

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1];
} BITMAPINFO;
```

Presentation Manager 1.2 DIB

```
typedef struct _BITMAPCOREINFO {
    BITMAPCOREHEADER    bmciHeader;
    RGBTRIPLE bmciColors[1];
} BITMAPCOREINFO;
```

These structures are essentially alike, and this section discusses them simultaneously. Each field name for the Windows BITMAPINFO structure is followed by the corresponding field name for the Presentation Manager BITMAPCOREINFO 1.2 structure, in parentheses.

The following table describes the fields.

Windows (PM) Field	Description
bmiHeader (**bmciHeader**)	Specifies information about the dimensions and color format of the DIB. The BITMAPINFOHEADER and BITMAPCOREHEADER data structures are described in the next section.
bmiColors (**bmciColors**)	Specifies the DIB color table. The RGBQUAD and RGBTRIPLE data structures are described in "Bitmap Color Table," later in this chapter.

Information Header Structures

The BITMAPINFOHEADER and BITMAPCOREHEADER structures contain information about the dimensions and color format of Windows 3.0 and Presentation Manager 1.2 DIBs, respectively. They are defined as follows:

Windows 3.0 DIB

```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    DWORD biWidth;
    DWORD biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    DWORD biXPelsPerMeter;
    DWORD biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
 } BITMAPINFOHEADER;
```

Presentation Manager 1.2 DIB

```
typedef struct tagBITMAPCOREHEADER {
    DWORD   bcSize;
    WORD    bcWidth;
    WORD    bcHeight;
    WORD    bcPlanes;
    WORD    bcBitCount;
 } BITMAPCOREHEADER;
```

Because these structures are essentially alike, except for the added fields in the Windows 3.0 structure, this section discusses them simultaneously. Each field name for the Windows structure is followed by the corresponding field name for the Presentation Manager structure, in parentheses.

Common Fields

The following fields are present in both the Windows 3.0 and Presentation Manager 1.2 formats:

Windows (PM) Field	Description
biSize (**bcSize**)	Specifies the number of bytes required by the BITMAPINFOHEADER structure. You can use this field to distinguish between Windows 3.0 and Presentation Manager 1.2 DIBs.
biWidth (**bcWidth**)	Specifies the width of the DIB in pixels.
biHeight (**bcHeight**)	Specifies the height of the DIB in pixels.
biPlanes (**bcPlanes**)	Specifies the number of planes for the target device. Must must be set to 1.
wBitCount (**bcBitCount**)	Specifies the number of bits-per-pixel. See " Interpreting the Color Table," later in this section, for more information.

Windows Fields

The following fields are present only in the Windows 3.0 BITMAPINFOHEADER structure:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>biCompression</td><td>Specifies the type of compression for a compressed bitmap. It can be one of the following values:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>BI_RGB</td><td>Specifies that the bitmap is not compressed.</td></tr>
<tr><td>BI_RLE4</td><td>Specifies a run-length encoded format for bitmaps with 4 bits-per-pixel. The compression format is a two-byte format consisting of a count byte followed by two word-length color indexes.</td></tr>
<tr><td>BI_RLE8</td><td>Specifies a run-length encoded format for bitmaps with 8 bits-per-pixel. The compression format is a two-byte format consisting of a count byte followed by a color-index byte.</td></tr>
</table>
See “Windows 3.0 Bitmap Compression Formats” later in this document for information about the encoding schemes.</td></tr>
<tr><td>biSizeImage</td><td>Specifies the size in bytes of the image.</td></tr>
<tr><td>biXPelsPerMeter</td><td>Specifies the horizontal resolution in pixels per meter of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.</td></tr>
<tr><td>biYPelsPerMeter</td><td>Specifies the vertical resolution in pixels per meter of the target device for the bitmap.</td></tr>
<tr><td>biClrUsed</td><td>Specifies the number of color values in the color table actually used by the bitmap. Possible values follow.
<table>
<tr><th>Value</th><th>Result</th></tr>
<tr><td>0</td><td>Bitmap uses the maximum number of colors corresponding to the value of the wBitCount field.</td></tr>
<tr><td>Nonzero</td><td>If the wBitCount value is less than 24, the biClrUsed value indicates the actual number of colors which the graphics engine or device driver will access.

If the wBitCount value is 24, the biClrUsed value indicates the size of the reference color table used to optimize performance of Windows color palettes.</td></tr>
</table>
If the bitmap is a “packed” bitmap (that is, a bitmap in which the bitmap array immediately follows the BITMAPINFO header and which is referenced by a single pointer), the biClrUsed field must be set to 0 or to the actual size of the color table. See “Interpreting the Color Table,” later in this section, for more information on how this field affects the interpretation of the color table.</td></tr>
<tr><td>biClrImportant</td><td>Specifies the number of color indexes that are considered important</td></tr>
</table>

for displaying the bitmap. If this value is 0, then all colors are important.

Bitmap Color Table

The color table is a collection of 24-bit RGB values. There are as many entries in the color table as there are colors in the bitmap. The color table isn't present for bitmaps with 24 color bits because each pixel is represented by 24-bit RGB values in the actual bitmap data area.

Color Table Structure

The color table for Windows 3.0 and Presentation Manager 1.2 DIBs consists of an array of RGBQUAD and RGBTRIPLE structures, respectively. These structures are defined as follows:

Windows 3.0 DIB

```
typedef struct tagRGBQUAD {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
 } RGBQUAD;
```

Presentation Manager 1.2 DIB

```
typedef struct tagRGBTRIPLE {
   BYTE rgbtBlue;
   BYTE rgbtGreen;
   BYTE rgbtRed;
} RGBTRIPLE;
```

Because these structures are essentially alike, this section discusses them simultaneously. Each field name for the Windows RGBQUAD structure is followed by the corresponding field name for the Presentation Manager RGBTRIPLE structure, in parentheses.

Order of Colors

The colors in the table should appear in order of importance. This can help a device driver render a bitmap on a device that cannot display as many colors as there are in the bitmap. If the DIB is in Windows 3.0 format, the driver can use the **biClrImportant** field of the BITMAPINFOHEADER structure to determine which colors are important.

Field Descriptions

The RGBQUAD (RGBTRIPLE) structure contains the following fields:

Windows (PM) Field	Description
rgbBlue (**rgbtBlue**)	Specifies the blue intensity.
rgbGreen (**rgbtGreen**)	Specifies the green intensity.
rgbRed (**rgbtRed**)	Specifies the red intensity.
rgbReserved (no PM equivalent)	Not used. Must be set to 0.

Locating the Color Table

An application can use the **biSize** (**bcSize**) field of the BITMAPINFOHEADER (BITMAPCOREHEADER) structure to locate the color table. Each of the following statements assigns the pColor variable the byte offset of the color table from the beginning of the file:

```
// Windows 3.0 DIB
pColor = (LPSTR)pBitmapInfo + (WORD)pBitmapInfo->biSize

// Presentation Manager 1.2 DIB
pColor = (LPSTR)pBitmapCoreInfo + (WORD)pBitmapCoreInfo->bcSize
```

Interpreting the Color Table

The **biSize** (**bcSize**) field of the BITMAPINFOHEADER (BITMAPCOREHEADER) structure specifies how many bits define each pixel and specifies the maximum number of colors in the bitmap. Its value affects your interpretation of the color table.

The **biSize** (**bcSize**) field can have any of the following values:

Value	Meaning
1	The 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	The bitmap has a maximum of 16 colors. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, then 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 16th table entry.
8	The bitmap has a maximum of 256 colors. Each pixel in the bitmap is represented by a byte-sized index into the color table. For example, if the first byte in the bitmap is 0x1F, then the first pixel has the color of the thirty-second table entry.
24	The bitmap has a maximum of 2^{24} colors. The **bmiColors** (**bmciColors**) field is NULL, and each three bytes in the bitmap array represent the relative intensities of red, green, and blue, respectively, of a pixel.

Note on Windows DIBs

For Windows 3.0 DIBs, the field of the BITMAPINFOHEADER structure specifies the number of color indexes in the color table actually used by the bitmap. If the **biClrUsed** field is set to 0, the bitmap uses the maximum number of colors corresponding to the value of the field.

Bitmap Data

The bits in the array are packed together, but each line of pixels, or scan line, must be zero-padded to end on a LONG boundary. When the bitmap is in memory, segment boundaries can appear anywhere in the bitmap. The origin of the bitmap is the lower-left corner. The following section discusses compression formats for the Windows 3.0 bitmap data.

Windows 3.0 Bitmap Compression Formats

Windows supports run-length encoded formats for compressing 4- and 8-bit bitmaps. Compression reduces the disk and memory storage required for the bitmap. The following sections describe the compression formats.

Compression of 8-Bit-Per-Pixel DIBs

When the **biCompression** field is set to BI_RLE8, the bitmap is compressed using a run-length encoding format for an 8-bit bitmap. This format uses two modes:

RDIB' form is defined as follows, using the standard RIFF form definition notation:

```
<RDIB-form> →       RIFF (  'RDIB' data( <DIB-data> ))
```

The <**DIB-data**> format is defined in "Device Independent Bitmap File Format," earlier in this chapter.

Extended RDIB Format

The extended RDIB format, designed to incorporate enhancements such as compression, is defined as follows:

```
<RDIB-form> →
          RIFF('RDIB'
                        <bmhd-ck>               // Bitmap header chunk
                        [ <pal-file> |          // Internal palette chunk
                          <XPAL-ck>   ]         // External palette chunk
                        <bitmap-data> )         // Bitmap data
```

The <**pal-file**> chunk can be any of the palette-file formats discussed in "Palette File Format," later in this chapter. The <**bmhd-ck**>, <**XPAL-chunk**>, and <**bitmap-data**> are described in the following sections.

Bitmap Header Chunk

The <**bmhd-ck**> bitmap header chunk is defined as follows:

```
<bmhd-chunk> → bmhd( struct {
      DWORD   dwMemSize;            // If dwPelFormat is 'data', only these
      DWORD   dwPelFormat;          // four fields are present.
      WORD    wTransType;
```

```
        DWORD   dwTransVal;
        DWORD   dwHdrSize;              // Fields from dwHdrSize forward match
        DWORD   dwWidth;                // the Windows BITMAPINFOHEADER
        DWORD   dwHeight;               // structure, though some fields can
        WORD    dwPlanes;               // contain new values.
        WORD    dwBitCount;
        DWORD   dwCompression;
        DWORD   dwSizeImage;
        DWORD   dwXPelsPerMeter;
        DWORD   dwYPelsPerMeter;
        DWORD   dwClrUsed;
        DWORD   dwClrImportant;
} )
```

If the **dwCompression** field equals BI_RGB or BI_RLE8 or BI_RLE4, then the extended RDIB has the same bitmap format as a simple RDIB.

Each pixel format defines the orientation, or position of the bitmap origin. Windows bitmaps (identified by a value of 'data' in the **dwPelFormat** field) have the origin at the bottom left. By default, the other formats have the origin at the top left.

Field	Description
dwMemSize	Equal to the size of the bitmap bits if the bits are uncompressed. For RDIBs with **dwPelFormat** equal to 'data,' **dwMemSize** has one of the following values:

Image Type	Field Value
Non-RLE	Same as **dwSizeImage** value
8-bit RLE	Size as an uncompressed, 8-bit image
4-bit RLE	Size as an uncompressed, 4-bit image

dwPelFormat — Specifies a FOURCC code defining the pixel format of the bitmap data. The bitmap data is stored in a chunk (or chunks) that has the same chunk ID as is contained in **dwPelFormat**. The compression scheme and pixel depth of the bitmap data are recorded in the **dwCompression** and **dwBitCount** fields. The current bitmap data values are as follows:

Value	Bitmap Data Location and Format
'data'	Bitmap data is stored in a 'data' chunk using the format defined for Windows 3.0 device independent bitmaps (DIBs). An application can display the bitmap properly even if the fields after (and including) **dwMemSize** are ignored.
'palb'	Bitmap data is stored in a 'palb' chunk. The pixel format is one of the Windows 3.0 RGB palettized formats (1 to 8 bpp, depending on the value of the **dwBitCount** field).
'rgbb'	Bitmap data is stored in a 'rgbb' chunk. Pixel format is packed, unpalettized RGB represented at 16, 24, or 32 bits per pixel. The following shows the ordering of the RGB bits for each pixel-depth value. The first extra bit (if present) is the high-order bit.

dwBitCount	Extra	Red	Green	Blue

15	1	5	5	5
16	0	5	6	5
24	0	8	8	8
32	8	8	8	8

'yuvb' Bitmap data is stored in a ‘yuvb’ chunk. Pixel format is packed, unpalettized YUV. The exact pixel format is currently undefined. By the time this draft is final, the pixel format will be defined similarly to the ‘rgbb’ definition.

wTransType

Specifies the type of transparency representation, if any, used for this image. This is normally used for either image overlay applications, where one image may be visually on top of another, and all pels of the transparency color should not be drawn. Examples include sprites, clip art and motion video overlay. Wherever the transparency color occurs in the picture, the background should be visible.

This information is stored with the image, so that multiple images that use the same color map may all have different transparency color. There are 5 different values for the transparency variable. These are:

Value	**Result**
BITT_NONE (0x0000)	No pels are considered transparent in this image.
BITT_MAPINDEX (0x0001)	One of the color map/palette entries should be considered the transparency color. All instances of this pel should NOT be drawn, and the existing background should be allowed to show through.
BITT_SINGLECOLOR (0x0002)	A single RGB or YUV value is considered transparent and should not be drawn.
BITT_BITPLANE (0x0003)	An individual bit plane is considered transparent, and all pels that have that bit or bits “on” should not be drawn.
BITT_MULTILEVEL (0x0004)	A set of bits indicate multiple levels of transparency or opacity. This is usually used with 32-bit RGB, where the high 8 bits indicate transparency.

dwTransVal

These bytes allow the image definition to indicate the exact information about the transparent color. The information is dependent on the value of the **wTransType** as follows:

wTransType	**dwTransVal Contents**
BITT_NONE	Not used.
BITT_MAPINDEX	Specifies a palette index, either 0 through 16 or 0 through 255, depending on the number of palette entries.
BITT_SINGLECOLOR	Specifies an RGB or YUV value (2 to 4 bytes in size, depending on the pixel format

	specified by **dwPelFormat**). All pels that match **dwTransVal** should be considered transparent.
BITT_BITPLANE	Specifies a bit mask identifying the bits used to indicate a transparent pel. Any pel that has this set of bits set is totally transparent. This allows multiple colors to be considered transparent. This method works for palettized images; in this case, the value refers to a map entry that is considered transparent.
BITT_MULTILEVEL	Specifies bits to use for transparency levels. These bits act as a mask on every pel, and each pel can be matched to the mask to determine the transparency level for the pel. For example, if **dwTransVal** has value 0xFF000000, then there are 256 levels of transparency. Each pel can be evaluated against the mask. If the pel has a value FFxxxxxx, then it is fully transparent. If the pel has a value 00xxxxxx, then it is fully visible. If the pel has a value 7Fxxxxxx, then the pel is half visible.

dwHdrSize Specifies the size of the data portion of the **<bmhdr>** chunk. This is always 40, the size of the BITMAPINFOHEADER structure.

dwWidth Specifies the width of the DIB in pixels.

dwHeight Specifies the height of the DIB in pixels.

wPlanes Specifies the number of planes. This value is normally 1, but it can be 3 or 4 for 24-bit RGB and 32-bit RGB images, respectively. In a multiplane DIB, each color component (for example, red, green, and blue) is stored as a separate plane, and each plan is stored in a separate bitmap data chunk. For example, in a 3-plane, 24-bit ‘rgbb’ bitmap, the red colors are stored in one ‘rgbb’ chunk, the green colors in a second ‘rgbb’ chunk, and the blue colors in a third ‘rgbb’ chunk.

Allowing the separate RGB planes to be compressed independently can dramatically improve the compression ratio. The **wPlanes** value must be 1 if **dwPelFormat** equals ‘data’.

wBitCount Specifies the number of bits per pixel. If the **dwPelFormat** field equals ‘data’, this field must contain values compatible with the Windows 3.0 DIB definition.

dwCompression Specifies the type of compression for a compressed bitmap. It can be one of the following values:

Value	**Meaning**
BI_NONE (0xFFFF0000)	Specifies that the bitmap is not compressed. Pixel values are *not* padded to four-byte boundaries.
BI_RGB (0x00000000)	Specifies that the bitmap is an uncompressed, 1-, 4-, 8-, or a 24-bit image. For 24-bit images, the palette is optional. Bitmap bits are represented as defined by Windows 3.0 for BI_RGB DIBs. The

	dwPelFormat field must be set to ‘data’.
BI_RLE8 (0x00000001)	Specifies a run-length encoded, compressed bitmap (as defined by Windows 3.0 BI_RLE8 DIBs). The palette is required. The **dwPelFormat** field must be set to ‘data’.
BI_RLE4 (0x00000002)	Specifies a run-length encoded, compressed bitmap (as defined by Windows 3.0 BI_RLE4 DIBs). The palette is required. The **dwPelFormat** field must be set to ‘data’.
BI_PACK (0xFFF0001)	Specifies a simple PACKBITS byte compression scheme consisting of one-byte counts followed by byte data, in the form: <count byte *n*><data byte1><data byte2>...<data byte *n*> <count byte *n*><data byte to repeat> The high-order bit of the count byte *n* is a decision bit:

***n* Value**	**Data Representation**
n < 0x80	A run of *n*+1 non-repeating bytes follows.
n > 0x80	Data byte is repeated (*n*-0x80+1) times.
n = 0x80	Reserved.

BI_TRANS (0xFFFF0002)	Specifies transitional compression, using a table of byte transitions or sequences. See “Transitional Compression,” following this table.
BI_CCC (0xFFFF0003)	Specifies CCC compression, a method involving encoding each 4-by-4 block of the image using two colors. See “CCC Compression,” following this table.
BI_JPEGN (0xFFFF0004)	To be defined later, when the ISO completes the official specification.

dwSizeImage Specifies the size in bytes of the compressed image.

dwXPelsPerMeter Specifies the horizontal resolution in pixels per meter of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device. This field is set to zero if unused.

dwYPelsPerMeter Specifies the vertical resolution in pixels per meter of the target device for the bitmap. This field is set to zero if unused.

dwClrUsed Specifies the number of palette entries actually used by the bitmap. Possible values follow.

Value	**Result**
0	Bitmap uses the maximum number of colors corresponding to the value of the **wBitCount** field.

	Nonzero	If the **wBitCount** is less than 24, **dwClrUsed** specifies the actual number of colors which the graphics engine or device driver will access. If the **wBitCount** field is set to 24, **dwClrUsed** specifies the size of the reference color table used to optimize performance of Windows color palettes.
dwClrImportant	Specifies the number of color indexes that are considered important for displaying the bitmap. If this value is 0, then all colors are important.	

Transitional Compression

If the **dwCompression** field is set to BI_TRANS, the data is transitionally compressed using a table of byte transitions or sequences. Values in the data indicate a table position to start at, and the table provides continuing references to other table positions. Transitional compression applies only to eight-bit data, either from an eight-bit palettized image or from a multi-plane image in which each color component is represented in eight bits.

The table consists of up to 256 16-byte rows at the beginning of the data section of the object. Nibbles (half-bytes) in the data section indicate an offset into a table row, at which location is stored the actual byte value. The actual value then becomes the row applicable to the next data nibble. The transitional encoding scheme is described more fully in a separate IBM document.

In transitional compression, the data section is a two-part compound object having the following items:

- A transition table
- The compressed image data

The transition table consists of an integer indicating the table size in bytes and a table of 16-byte rows. The first byte in each row is a row number and the next 15 are transition values. Rows are in descending sequence. The image is compressed according to the following rules:

- Data is in nibbles (half-bytes) or in nibble-pairs (successive half-bytes which may cross a byte boundary).
- The first byte is a nibble-pair. It is the first byte of the image and also the first row number.
- Following a nibble-pair is a series of transition nibbles (1-15) ended by a terminator (0). Each transition nibble indicates an offset in the current row at which the next byte in the image is found; this value is also the next row number.
- The terminator indicates that the next image byte is not in the table, but instead in the following nibble-pair. This value is also the next row number.
- If the picture has an odd number of nibbles (i.e., it ends in the first half of the last byte), an extra zero nibble is included.

CCC Compression

TBD.

Palette Chunk

A PLT chunk represents a color table and consists of a valid PAL file. The PAL file format is defined in "Palette File Format," later in this chapter.

External Palette Chunk

Instead of a PLT chunk, an RDIB may contain an XPLT chunk, which indicates that the bitmap's palette is stored outside the bitmap. The palette might be stored in a separate file or as a separate compound file element. The XPLT chunk indicates the name and location of the external palette chunk and is defined as follows:

```
<XPLT-chunk> → XPLT(<fccLocation:FOURCC> <szPaletteName::ZSTR>)
```

The **fccLocation** contains one of the following FOURCC values specifying the location of the external chunk:

fccLocation Value	Chunk Location
'full'	Palette is located in an external file, and the **szPaletteName** value specifies a complete filename with path.
'file'	Palette is located in an external file, and the **szPaletteName** value specifies a filename without path.
'elem'	Palette is located in the same compound file containing the DIB. The **szPaletteName** value specifies the name of the compound file element.

The **szPaletteName** consists of a null-terminated string (ZSTR) containing the name of the external chunk containing the palette.

Bitmap Data Chunk

The <bitmap-data> contains bitmap data in the format specified by the **biPelFormat** field of the <bmhd-chunk>.

MIDI and RIFF MIDI File Formats

The Musical Instrument Digital Interface (MIDI) file format represents a Standard MIDI File, as defined by the MIDI Manufacturers Association. A MIDI file contains commands instructing instruments to play specific notes and perform other operations.

The specifications for MIDI and MIDI files can be obtained from the following organization:

International MIDI Association (IMA)
5316 W. 57th Street
Los Angeles, CA 90056
(213) 649-6434.

The 'RMID' format consists of a standard MIDI file enclosed in a RIFF chunk. Enclosing the MIDI file in a 'RIFF' chunk allows the file to be consistently identified; for example, an 'INFO' list can be included in the file.

The 'RMID' form is defined as follows, using the standard RIFF form definition:

```
<RMID-form>   →  RIFF ('RMID' data( <MIDI-data> ))
```

The <MIDI-data> is equivalent to a Standard MIDI File.

Palette File Format

The Palette (PAL) File Format represents a logical palette, which is a collection of colors represented as RGB values. There are two types of PAL formats:

- A simple PAL format
- An extended PAL format

Simple PAL Format

The simple PAL format is defined as follows:

```
RIFF('PAL' data( <palette:LOGPALETTE> ))
```

LOGPALETTE is the Windows 3.0 logical palette structure, defined as follows:

```
typedef struct tagLOGPALETTE {
  WORD            palVersion;
  WORD            palNumEntries;
  PALETTEENTRY        palPalEntry[];
} LOGPALETTE;
```

The LOGPALETTE structure fields are as follows:

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.

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;
```

The PALETTEENTRY structure fields are as follows:

the bitmap's palette is stored outside the bitmap. The palette might be stored in a separate file or as a separate compound file element. The XPLT chunk indicates the name and location of the external palette chunk and is defined as follows:

```
<XPLT-chunk> → XPLT(<fccLocation:FOURCC> <szPaletteName::ZSTR>)
```

The **fccLocation** contains one of the following FOURCC values specifying the location of the external chunk:

fccLocation Value	Chunk Location
'full'	Palette is located in an external file, and the **szPaletteName** value specifies a complete filename with path.
'file'	Palette is located in an external file, and the **szPaletteName** value specifies a filename without path.
'elem'	Palette is located in the same compound file containing the DIB. The **szPaletteName** value specifies the name of the compound file element.

The **szPaletteName** consists of a null-terminated string (ZSTR) containing the name of the external chunk containing the palette.

Bitmap Data Chunk

The <bitmap-data> contains bitmap data in the format specified by the **biPelFormat** field of the <bmhd-chunk>.

MIDI and RIFF MIDI File Formats

The Musical Instrument Digital Interface (MIDI) file format represents a Standard MIDI File, as defined by the MIDI Manufacturers Association. A MIDI file contains commands instructing instruments to play specific notes and perform other operations.

The specifications for MIDI and MIDI files can be obtained from the following organization:

International MIDI Association (IMA)
5316 W. 57th Street
Los Angeles, CA 90056
(213) 649-6434.

The 'RMID' format consists of a standard MIDI file enclosed in a RIFF chunk. Enclosing the MIDI file in a 'RIFF' chunk allows the file to be consistently identified; for example, an 'INFO' list can be included in the file.

The 'RMID' form is defined as follows, using the standard RIFF form definition:

```
<RMID-form>   →  RIFF ('RMID' data( <MIDI-data> ))
```

The <MIDI-data> is equivalent to a Standard MIDI File.

Palette File Format

The Palette (PAL) File Format represents a logical palette, which is a collection of colors represented as RGB values. There are two types of PAL formats:

- A simple PAL format
- An extended PAL format

Simple PAL Format

The simple PAL format is defined as follows:

```
RIFF('PAL' data( <palette:LOGPALETTE> ))
```

LOGPALETTE is the Windows 3.0 logical palette structure, defined as follows:

```
typedef struct tagLOGPALETTE {
  WORD            palVersion;
  WORD            palNumEntries;
  PALETTEENTRY        palPalEntry[];
} LOGPALETTE;
```

The LOGPALETTE structure fields are as follows:

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.

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;
```

The PALETTEENTRY structure fields are as follows:

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.

Extended PAL Format

The extended PAL format includes the following:

- A palette-header chunk
- A data chunk containing an RGB palette (consisting of a LOGPALETTE structure) or some other palette type, including YUV and XYZ palettes.

For an RGB palette, the extended PAL format is represented as follows:

```
RIFF('PAL' plth( <palette-header> ) data( <LOGPALETTE-data> ))
```

For a YUV palette, the extended PAL format is represented as follows:

```
RIFF('PAL' plth( <palette-header> ) yuvp( <YUV-LOGPALETTE-data> ))
```

Both the **<LOGPALETTE-data>** and **<YUV-LOGPALETTE-data>** use the Windows 3.0 LOGPALETTE structure, described in " Simple PAL Format," earlier in this section. The **<YUV-LOGPALETTE-data>** contains YUV values instead of RGB values.

The 'plth' chunk is defined as follows:

```
<plth-ck> → PLT( struct {
          DWORD   dwMapType;
          WORD    wWhite;             // Fields from this point on are
          WORD    wBlack;             // optional. If they are included
          WORD    wBorder;            // but not used, set them to 0xFFFF.
          WORD    wRegisteredMap;
          WORD    wCustomBase;        // If an application encounters a
          WORD    wCustomCnt;         // 'PLT' chunk smaller than shown
          WORD    wRsvBase;           // here, it should treat the missing
          WORD    wRsvCount;          // fields as unused.
          WORD    wArtBase;
          WORD    wArtCnt;
          WORD    wNumIntense;
} )
```

The structure fields are described in the following:

Field	Description

dwMapType

FOURCC code specifying the type of palette. Currently, the following palette types are identified:

Code	Description
‘data’	Specifies an RGB palette. Data chunk contains a LOGPALETTE structure.
‘yuvp’	Specifies a YUV palette. Data chunk contains a YUV palette.
‘xyzp’	Specifies an XYZ palette. Data chunk contains a XYZ palette.

wWhite
wBlack

Specify palette-map indices corresponding to the closest value of white and black. These identify the pair of colors with the best contrast for use in cursors, calibration, etc. These values are usually changed if the palette changes. Ignore these fields if they contain 0xFFFF.

wBorder

Specifies the index of the palette entry to be used for any display-border regions, if supported by the display device. Ignore this field if it contains 0xFFFF.

wRegisteredMap

Specifies how many palette entries correspond to a registered color map. Registered entries are stored at the front of the palette. Ignore this field if it contains 0xFFFF.

Registered map entries are always stored at the beginning of the palette, so **wRegisteredMap** also indicates the index of the first custom color in the palette. Registered color maps include predefined palettes for general use, forest/nature, or seasides. Currently defined values are the following:

Value	Description
PAL_UNREGISTERED (0xFFFF)	Color map does not contain colors from a registered color map.
PAL_VGA (0x0000)	Color map contains the standard 16 VGA colors.
PAL_AVC198 (0x0001)	

	Standard AVC 198-entry palette.
wCustomBase	Specifies the index of the first custom color of the palette. The beginning of the palette contains the entries of the registered map, so **wCustomBase** also indicates the number of entries in the registered palette. Map entries starting with **wCustomBase** comprise additional custom colors used in the bitmap. Ignore this value if **wRegisteredPalette** is PAL_UNREGISTERED, or if **wCustomBase** contains 0xFFFF.
wCustomCnt	Specifies the number of custom colors in the palette. Ignore this value if **wRegisteredPalette** is PAL_UNREGISTERED, or if this field contains 0xFFFF.
wRsvBase	Specifies the index of the first reserved color of the palette. Reserved colors are those reserved for menus, text, and other screen elements. Reserved colors must be stored contiguously. Ignore this field if it contains 0xFFFF.
wRsvCnt	Specifies the number of reserved entries. Ignore this field if it contains 0xFFFF.
wArtBase	Specifies the index of the first art color of the palette. Art colors are colors used for text and drawing. Art colors consist of a number of hues, each of which has multiple intensities. The various intensities are used for anti-aliasing, a method of using different shades of a color to improve the quality of images displayed on low-resolution devices. For example, if the first art color is red anti-aliased to black with three intensities, the first three entries in the palette would be dark red, medium red, and bright red. The art colors constitute an array, and all hues have the same number of intensities. The user can set both the number of hues and the number of intensities. Ignore these fields if they contain 0xFFFF.
wArtCnt	Specifies the number of art colors. Ignore this field if it contains 0xFFFF.
wNumIntense	Specifies the number of palette entries reserved for the anti-aliased levels of a given art color. This field must be present if **wArtBase** is present. Ignore this field if it contains 0xFFFF.

Rich Text Format (RTF)

The Rich Text Format (RTF) is a standard method of encoding formatted text and graphics using only 7-bit ASCII characters. Formatting includes different font sizes, faces, and styles, as well as paragraph alignment, justification, and tab control.

RTF is described in the *Microsoft Word Technical Reference: For Windows and OS/2*, published by Microsoft Press.

Waveform Audio File Format (WAVE)

This section describes the Waveform format, which is used to represent digitized sound.

The WAVE form is defined as follows. Programs must expect (and ignore) any unknown chunks encountered, as with all RIFF forms. However, **<fmt-ck>** must always occur before **<wave-data>**, and both of these chunks are mandatory in a WAVE file.

```
<WAVE-form> →
          RIFF( 'WAVE'
                  <fmt-ck>                // Format
                  [<fact-ck>]             // Fact chunk
                  [<cue-ck>]              // Cue points
                  [<playlist-ck>]         // Playlist
                  [<assoc-data-list>]     // Associated data list
                  <wave-data>  )          // Wave data
```

The WAVE chunks are described in the following sections.

WAVE Format Chunk

The WAVE format chunk **<fmt-ck>** specifies the format of the **<wave-data>**. The **<fmt-ck>** is defined as follows:

```
<fmt-ck> →       fmt(    <common-fields>
                         <format-specific-fields> )

<common-fields> →
          struct
          {
              WORD    wFormatTag;          // Format category
              WORD    wChannels;           // Number of channels
              DWORD   dwSamplesPerSec;     // Sampling rate
              DWORD   dwAvgBytesPerSec;    // For buffer estimation
              WORD    wBlockAlign;         // Data block size
          }
```

The fields in the **<common-fields>** chunk are as follows:

Field	Description
wFormatTag	A number indicating the WAVE format category of the file. The content of the **<format-specific-fields>** portion of the 'fmt' chunk, and the interpretation of the waveform data, depend on this value. You must register any new WAVE format categories. See "Registering Multimedia Formats" in Chapter 1, "Overview of Multimedia Specifications," for information on registering WAVE format categories. "Wave Format Categories," following this section, lists the currently defined WAVE format categories.
wChannels	The number of channels represented in the waveform data, such as 1 for mono or 2 for stereo.
dwSamplesPerSec	The sampling rate (in samples per second) at which each channel should be played.

dwAvgBytesPerSec	The average number of bytes per second at which the waveform data should be transferred. Playback software can estimate the buffer size using this value.
wBlockAlign	The block alignment (in bytes) of the waveform data. Playback software needs to process a multiple of **wBlockAlign** bytes of data at a time, so the value of **wBlockAlign** can be used for buffer alignment.

The **<format-specific-fields>** consists of zero or more bytes of parameters. Which parameters occur depends on the WAVE format category–see the following section for details. Playback software should be written to allow for (and ignore) any unknown **<format-specific-fields>** parameters that occur at the end of this field.

WAVE Format Categories

The format category of a WAVE file is specified by the value of the **wFormatTag** field of the 'fmt' chunk. The representation of data in **<wave-data>**, and the content of the **<format-specific-fields>** of the 'fmt' chunk, depend on the format category.

The currently defined open non-proprietary WAVE format categories are as follows:

wFormatTag Value	Format Category
WAVE_FORMAT_PCM (0x0001)	Microsoft Pulse Code Modulation (PCM) format

The following are the registered proprietary WAVE format categories:

wFormatTag Value	Format Category
IBM_FORMAT_MULAW (0x0101)	IBM mu-law format
IBM_FORMAT_ALAW (0x0102)	IBM a-law format
IBM_FORMAT_ADPCM (0x0103)	IBM AVC Adaptive Differential Pulse Code Modulation format

The following sections describe the Microsoft WAVE_FORMAT_PCM format.

Pulse Code Modulation (PCM) Format

If the **wFormatTag** field of the **<fmt-ck>** is set to WAVE_FORMAT_PCM, then the waveform data consists of samples represented in pulse code modulation (PCM) format. For PCM waveform data, the **<format-specific-fields>** is defined as follows:

```
<PCM-format-specific> →
          struct
          {
              WORD    wBitsPerSample;         // Sample size
          }
```

The **wBitsPerSample** field specifies the number of bits of data used to represent each sample of each channel. If there are multiple channels, the sample size is the same for each channel.

For PCM data, the **wAvgBytesPerSec** field of the 'fmt' chunk should be equal to the following formula rounded up to the next whole number:

$$\text{wChannels} \times \text{wBitsPerSecond} \times \frac{\text{wBitsPerSample}}{8}$$

The **wBlockAlign** field should be equal to the following formula, rounded to the next whole number:

$$\text{wChannels} \times \frac{\text{wBitsPerSample}}{8}$$

Data Packing for PCM WAVE Files

In a single-channel WAVE file, samples are stored consecutively. For stereo WAVE files, channel 0 represents the left channel, and channel 1 represents the right channel. The speaker position mapping for more than two channels is currently undefined. In multiple-channel WAVE files, samples are interleaved.

The following diagrams 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 0 (right)

Data Packing for 8-Bit Stereo PCM

The following diagrams 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

Data Format of the Samples

Each sample is contained in an integer *i*. The size of *i* is the smallest number of bytes required to contain the specified sample size. The least significant byte is stored first. The bits that represent the sample amplitude are stored in the most significant bits of *i*, and the remaining bits are set to zero.

For example, if the sample size (recorded in **nBitsPerSample**) is 12 bits, then each sample is stored in a two-byte integer. The least significant four bits of the first (least significant) byte is set to zero.

The data format and maximum and minimums values for PCM waveform samples of various sizes are as follows:

Sample Size	Data Format	Maximum Value	Minimum Value
One to eight bits	Unsigned integer	255 (0xFF)	0
Nine or more bits	Signed integer *i*	Largest positive value of *i*	Most negative value of *i*

For example, the maximum, minimum, and midpoint values for 8-bit and 16-bit PCM waveform data are as follows:

Format	Maximum Value	Minimum Value	Midpoint Value
8-bit PCM	255 (0xFF)	0	128 (0x80)
16-bit PCM	32767 (0x7FFF)	-32768 (-0x8000)	0

Examples of PCM WAVE Files

Example of a PCM WAVE file with 11.025 kHz sampling rate, mono, 8 bits per sample:

```
RIFF( 'WAVE'      fmt(1, 1, 11025, 11025, 1, 8)
                      data( <wave-data> ) )
```

Example of a PCM WAVE file with 22.05 kHz sampling rate, stereo, 8 bits per sample:

```
RIFF( 'WAVE'      fmt(1, 2, 22050, 44100, 2, 8)
                      data( <wave-data> ) )
```

Example of a PCM WAVE file with 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> ) )
```

Storage of WAVE Data

The **<wave-data>** contains the waveform data. It is defined as follows:

```
<wave-data> →       { <data-ck> | <data-list> }

<data-ck> →     data( <wave-data> )

<wave-list> →       LIST( 'wavl' {     <data-ck> |              // Wave
samples
                                       <silence-ck> }... )      // Silence

<silence-ck> →      slnt( <dwSamples:DWORD> )           // Count of
                                                        // silent samples
```

Note: The 'slnt' chunk represents silence, not necessarily a repeated zero volume or baseline sample. In 16-bit PCM data, if the last sample value played before the silence section is a 10000, then if data is still output to the D to A converter, it must maintain the 10000 value. If a zero value is used, a click may be heard at the start and end of the silence section. If play begins at a silence section, then a zero value might be used since no other information is available. A click might be created if the data following the silent section starts with a nonzero value.

FACT Chunk

The **<fact-ck>** fact chunk stores important information about the contents of the WAVE file. This chunk is defined as follows:

```
<fact-ck> → fact( <dwFileSize:DWORD> )          // Number of samples
```

The “fact” chunk is required if the waveform data is contained in a “wavl” LIST chunk and for all compressed audio formats. The chunk is not required for PCM files using the “data” chunk format.

The "fact" chunk will be expanded to include any other information required by future WAVE formats. Added fields will appear following the **<dwFileSize>** field. Applications can use the chunk size field to determine which fields are present.

Cue-Points Chunk

The **<cue-ck>** cue-points chunk identifies a series of positions in the waveform data stream. The **<cue-ck>** is defined as follows:

```
<cue-ck> →      cue(    <dwCuePoints:DWORD>             // Count of cue points
                            <cue-point>... )            // Cue-point table

<cue-point> →       struct {
                        DWORD  dwName;
                        DWORD  dwPosition;
                        FOURCC fccChunk;
                        DWORD  dwChunkStart;
                        DWORD  dwBlockStart;
                        DWORD  dwSampleOffset;
                    }
```

The **<cue-point>** fields are as follows:

Field	Description
dwName	Specifies the cue point name. Each **<cue-point>** record must have a unique **dwName** field.
dwPosition	Specifies the sample position of the cue point. This is the sequential sample number within the play order. See “Playlist Chunk,” later in this document, for a discussion of the play order.
fccChunk	Specifies the name or chunk ID of the chunk containing the cue point.
dwChunkStart	Specifies the file position of the start of the chunk containing the cue point. This is a byte offset relative to the start of the data section of the ‘wavl’ LIST chunk.
dwBlockStart	Specifies the file position of the start of the block containing the position. This is a byte offset relative to the start of the data section of the ‘wavl’ LIST chunk.
dwSampleOffset	Specifies the sample offset of the cue point relative to the start of the block.

Examples of File Position Values

The following table describes the **<cue-point>** field values for a WAVE file containing multiple 'data' and 'slnt' chunks enclosed in a 'wavl' LIST chunk:

Cue Point Location	Field	Value
In a 'slnt' chunk	**fccChunk**	FOURCC value 'slnt'.
	dwChunkStart	File position of the 'slnt' chunk relative to the start of the data section in the 'wavl' LIST chunk.
	dwBlockStart	File position of the data section of the 'slnt' chunk relative to the start of the data section of the 'wavl' LIST chunk.
	dwSampleOffset	Sample position of the cue point relative to the start of the 'slnt' chunk.
In a PCM 'data' chunk	**fccChunk**	FOURCC value 'data'.
	dwChunkStart	File position of the 'data' chunk relative to the start of the data section in the 'wavl' LIST chunk.
	dwBlockStart	File position of the cue point relative to the start of the data section of the 'wavl' LIST chunk.
	dwSampleOffset	Zero value.
In a compressed 'data' chunk	**fccChunk**	FOURCC value 'data'.
	dwChunkStart	File position of the start of the 'data' chunk relative to the start of the data section of the 'wavl' LIST chunk.
	dwBlockStart	File position of the enclosing block relative to the start of the data section of the 'wavl' LIST chunk. The software can begin the decompression at this point.
	dwSampleOffset	Sample position of the cue point relative to the start of the block.

The following table describes the **<cue-point>** field values for a WAVE file containing a single 'data' chunk:

Cue Point Location	Field	Value
Within PCM data	**fccChunk**	FOURCC value 'data'.
	dwChunkStart	Zero value.
	dwBlockStart	Zero value.
	dwSampleOffset	Sample position of the cue point relative to

		the start of the ‘data’ chunk.
In a compressed ‘data’ chunk	**fccChunk**	FOURCC value ‘data’.
	dwChunkStart	Zero value.
	dwBlockStart	File position of the enclosing block relative to the start of the ‘data’ chunk. The software can begin the decompression at this point.
	dwSampleOffset	Sample position of the cue point relative to the start of the block.

Playlist Chunk

The **<playlist-ck>** playlist chunk specifies a play order for a series of cue points. The **<playlist-ck>** is defined as follows:

```
<playlist-ck> →     plst(
                        <dwSegments:DWORD>        // Count of play segments
                        <play-segment>... )       // Play-segment table

<play-segment> →    struct {
                            DWORD dwName;
                            DWORD dwLength;
                            DWORD dwLoops;
                        }
```

The **<play-segment>** fields are as follows:

Field	Description
dwName	Specifies the cue point name. This value must match one of the names listed in the **<cue-ck>** cue-point table.
dwLength	Specifies the length of the section in samples.
dwLoops	Specifies the number of times to play the section.

Associated Data Chunk

The **<assoc-data-list>** associated data list provides the ability to attach information like labels to sections of the waveform data stream. The **<assoc-data-list>** is defined as follows:

```
<assoc-data-list> →     LIST(   'adtl'
                                    <labl-ck>                   // Label
                                    <note-ck>                   // Note
                                    <ltxt-ck>                   // Text with
data length
                                    <file-ck> )                 // Media file

<labl-ck> →             labl(   <dwName:DWORD>
                                    <data:ZSTR> )

<note-ck> →             note(   <dwName:DWORD>
                                    <data:ZSTR> )
```

```
<ltxt-ck> →          ltxt(   <dwName:DWORD>
                             <dwSampleLength:DWORD>
                             <dwPurpose:DWORD>
                             <wCountry:WORD>
                             <wLanguage:WORD>
                             <wDialect:WORD>
                             <wCodePage:WORD>
                             <data:BYTE>... )

<file-ck> →          file(   <dwName:DWORD>
                             <dwMedType:DWORD>
                             <fileData:BYTE>...)
```

Label and Note Information

The 'labl' and 'note' chunks have similar fields. The 'labl' chunk contains a label, or title, to associate with a cue point. The 'note' chunk contains comment text for a cue point. The fields are as follows:

Field	Description
dwName	Specifies the cue point name. This value must match one of the names listed in the <cue-ck> cue-point table.
data	Specifies a NULL-terminated string containing a text label (for the 'labl' chunk) or comment text (for the 'note' chunk).

Text with Data Length Information

The "ltxt" chunk contains text that is associated with a data segment of specific length. The chunk fields are as follows:

Field	Description
dwName	Specifies the cue point name. This value must match one of the names listed in the <cue-ck> cue-point table.
dwSampleLength	Specifies the number of samples in the segment of waveform data.
dwPurpose	Specifies the type or purpose of the text. For example, **dwPurpose** can specify a FOURCC code like 'scrp' for script text or 'capt' for close-caption text.
wCountry	Specifies the country code for the text. See "Country Codes" in Chapter 2, "Resource Interchange File Format," for a current list of country codes.
wLanguage, **wDialect**	Specify the language and dialect codes for the text. See "Language and Dialect Codes" in Chapter 2, "Resource Interchange File Format," for a current list of language and dialect codes.
wCodePage	Specifies the code page for the text.

Embedded File Information

The ‘file’ chunk contains information described in other file formats (for example, an ‘RDIB’ file or an ASCII text file). The chunk fields are as follows:

Field	Description
dwName	Specifies the cue point name. This value must match one of the names listed in the <cue-ck> cue-point table.
dwMedType	Specifies the file type contained in the **fileData** field. If the **fileData** section contains a RIFF form, the **dwMedType** field is the same as the RIFF form type for the file. This field can contain a zero value.
fileData	Contains the media file.

Chapter 4

Media Control Interface

The Media Control Interface (MCI) is a high-level command control 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 multimedia device, including audio playback and recording, as well as videodisc and videotape players.

MCI provides a standard command set for playing and recording multimedia devices and resource files. Developers creating multimedia applications are encouraged to use this high-level command interface rather than the low-level functions specific to each platform. The MCI command set acts as a platform-independent layer that sits between multimedia applications and the underlying system software.

The command set is extensible in two ways:

- Developers can incorporate new multimedia devices and file formats in the MCI command set by creating new MCI drivers to interpret the commands.
- New commands and command options can be added to support special features or functions required by new multimedia devices or file formats.

MCI Command Strings

Using MCI, an application can control multimedia devices using simple command strings like open, play, and close. The MCI commands provide a generic interface to different multimedia devices, reducing the number of commands a developer needs to learn. A multimedia application might even accept MCI commands from an end user and pass them unchanged to the MCI driver, which parses the command and performs the appropriate action.

A set of basic commands is supported by all MCI devices. Developers can also define MCI commands and command options specific to a particular multimedia device or file format. These device-specific commands and command options are needed only when the basic command set does not support a feature specific to the device or file format.

Example of MCI Command Use

The following example shows a series of MCI commands that play track 6 of an audio compact disc:

```
open cdaudio
set cdaudio time format tmsf
play cdaudio from 6 to 7
close cdaudio
```

The next example shows a similar series of MCI commands that play the first 10,000 samples of a waveform audio file:

```
open c:\mmdata\purplefi.wav type waveaudio alias finch
set finch time format samples
play finch from 1 to 10000 wait
close finch
```

Notice the following:

- The same basic commands (**open**, **play**, and **close**) are used with both devices.
- The **open** command for the "waveaudio" device includes a filename specification. The "waveaudio" device is a compound device (one associated with a media element), while the "cdaudio" device is a simple device (one without an associated media element).
- The **set** commands both specify time formats, but the time format options for the "cdaudio" device are different from those used with the "waveaudio" device.
- The parameters used with the **from** and **to** flags are appropriate to the respective device. For the "cdaudio" device, the parameters specify a range of tracks; for the "waveaudio" device, the parameters specify a range of samples.

Categories of MCI Command Strings

MCI command strings divide into the following categories:

- *System commands* are interpreted directly by MCI rather than being relayed to a device.
- *Required commands* are recognized by all MCI devices. If a device does not support a required command, it can return "unsupported function" in response to the message.
- *Basic commands* are optional commands. If a device uses a basic command, it must respond to all options for that command. If a device does not use a basic command, it can return "unrecognized command" in response to the message.
- *Extended commands* are specific to a device type or device class; for example, videodisc players. These commands contain both unique commands and extensions to the required and basic commands.

Command Syntax Conventions

This chapter uses the following documentation conventions:

Convention	Description
bold	MCI command or flag keyword.
italics	Command parameter to be replaced with a valid string, number, or rectangle specification.
" quotes"	Parameter text to be typed exactly as shown.
[brackets]	Optional flags or parameters

System Commands

The following list summarizes the system commands. MCI supports these commands directly rather than passing them to MCI devices.

Message	Description
sound	Play system sounds defined in a system setup file.
sysinfo	Returns information about MCI devices.

Required Commands

The following list summarizes the required commands. All devices recognize these messages. If a device does not support a required command, it can return " unsupported function" in response to the message.

Message	Description
capability	Obtains the capabilities of a device.
close	Closes the device.
info	Obtains textual information from a device.
open	Initializes the device.
status	Returns various status information from the device.

Basic Commands

The following list summarizes the basic commands. MCI devices are not required to recognize these commands. If the device does not recognize a basic command, it can return " unrecognized command" in response to the message.

Message	Description
load	Recalls data from a disk file.
pause	Stops playing.
play	Starts transmitting output data.
record	Starts recording input data.
resume	Resumes playing or recording from a paused state.
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 flags for this command supplement the flags for the command in the required command group.)
stop	Stops playing.

Extended Commands

MCI devices can have additional commands or extend the definition of the required and basic commands. While some extended commands only apply to a specific device driver most of them apply to all devices of a particular type. For example, the MIDI sequencer command set extends the **set** command to add time formats needed by MIDI sequencers.You can find descriptions of extended commands in the command tables in this chapter.

Extended Commands Reserved for Future Use

The following commands can be defined as extended commands. With the exception of the **delete** command, they are not currently defined for any MCI devices.

Message	Description
copy	Copies data to the Clipboard. Parameters and flags for this message vary according to the selected device.
cut	Moves data from the MCI element to the Clipboard. Parameters and flags for this message vary according to the selected device.
delete	Removes data from the MCI element. Parameters and flags for this message vary according to the selected device.

Creating a Command String

There are three components associated with each command string: the command, the name or ID of the device receiving the command, and the command arguments. A command string has the following form:

command device_name arguments

These components contain the following information:

- The *command* includes a command from the system, required, basic, or extended command set. Examples of commands include **open**, **close**, and **play**.

- The *device_name* designates the target of the *command*. MCI accepts the names of MCI device types and names of media elements for the *device_name*. An example of a device name is "cdAudio".

- The *arguments* specify the flags and parameters used by the *command*. Flags are key words recognized by the MCI command, and parameters are variables associated with the MCI command or flag. Parameters specify variable data values such as filenames, track or frame numbers, or speed values. You can use the following data types for the parameters in a string command:

    - Strings–String data types can be delimited by leading and trailing white space or by matching quotation marks. If MCI encounters a single (unmatched) quotation mark, it ignores the quotation mark. To embed a quote in string, use two quotes (""). To specify an empty string, you can use double quotes ("") for the string.

    - Signed long integers–Signed long integer data types are delimited by leading and trailing white space. Unless otherwise specified, integers can be positive or negative. If using negative integers, do not embed white space between the negative sign and the first digit.

    - Rectangle–Rectangle data types are an ordered list of four signed integer values. White space delimits this data type as well as separates each integer in the list.

    For example, the **play** command uses the arguments "**from** *position* **to** *position*" to specify starting and ending points for the playback. The **from** and **to** arguments are flags, and the two *position* values are parameters.

For example, the following command string instructs the CD audio player "cdaudio" to play from the start of the waveform to position 500:

```
play cdaudio from 0 to 500
```

Unspecified command arguments assume a default value. For example, if the flag *from* was unspecified in the previous example, the audio player would start playing at the current position.

About MCI Device Types

Your application identifies an MCI device by specifying an MCI *device type*. A device type indicates the physical type of device. The following table lists the currently defined MCI device types:

Device Type	Description
cdaudio[1]	CD audio player
dat	Digital audio tape player
digitalvideo	Digital video in a window (not GDI based)
other	Undefined MCI device
scanner	Image scanner
sequencer[1]	MIDI sequencer
vcr	Videotape recorder or player
videodisc[1]	Videodisc player
waveaudio[1]	Audio device that plays digitized waveform files

[1]An extended command set is provided for these devices.

If you have a particular device type installed more than once, the device type names in the system setup file have integers appended to them. This creates unique names for each MCI device type entry. For example, if the "cdaudio" device type is installed twice, the names "cdaudio1" and "cdaudio2" are used to create unique names for each occurrence of the device type. Each name usually refers to a different CD audio player in the system.

Using MCI Command Strings

The tables at the end of this chapter describe command strings for the MCI devices. The following sections describe commonly used command strings.

Opening a Device

Before using a device, you must initialize it with the **open** command. The number of devices you can have open depends on the amount of available memory. The **open** command has the following syntax:

open *device_name* [**shareable**] [**type** *device_type*] [**alias** *alias*]

The parameters for the **open** command are:

Parameters	Description

device_name	Specifies the destination device or MCI element name (filename).
shareable	Allows applications to share a common device or device element.
type *device_type*	Specifies the device when the *device_name* refers to an MCI element.
alias *alias*	Specifies an alternate name for the device.

MCI classifies device drivers as *compound* and *simple*. 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 file elements, the element name references a file and its path.

Simple device drivers do not require a device element for playback. For example, compact disc audio device drivers are simple device drivers.

Opening Simple Devices

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 type obtained from the system setup file. For example, you can open a videodisc device with the following command:

```
open videodisc1
```

Opening Compound Devices

There are three ways to open a compound device:

- By specifying just the device type
- By specifying both the element name and the device type
- By specifying just the element name

To determine the capabilities of a device, you can open a device by specifying only the device type. When opened this way, most compound devices will let you determine their capabilities and close them. For example, you can open the sequencer with the following command:

```
open sequencer
```

To associate a device element with a particular device, you must specify the element name and device type. In the **open** command, substitute the element name for the *device_name*, add the **type** flag, and substitute the name of the device you want to use for *device_type*. This combination lets your application specify the MCI device it needs to use. For example, you can open a device element of the waveaudio device with the following command:

```
open  right.wav  type  waveaudio
```

To associate a default MCI device with a device element, you can specify just an element name. In this case, MCI uses the filename extension of the element name to select the device type.

Using the Shareable Flag

The **shareable** flag lets multiple applications or tasks concurrently access the same device (or element) and device instance.

If your application opens a device or device element without the **shareable** flag, no other application can access it simultaneously. If your application opens a device or device element as shareable, other applications can also access it by also 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 that a device or device element is opened as shareable, a unique device ID is returned (even though the device IDs refer to the same instance)

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.

If a device can service only one application or task it will fail an open with the **shareable** flag.

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 open device. Each file element opened within a task must have a unique name. The **alias** flag described in the next section lets you use a unique name for each element.

Using the Alias Flag

The **alias** flag specifies an alternate name for the given device. The alias provides a shorthand notation for compound devices with lengthy pathnames. If your application creates a device alias, it must use the alias rather than the device name for all subsequent references.

Opening New Device Elements

To create a new device element for a task such as capturing a sound using waveform recording, specify **new** as a *device_name*. MCI does not save a new file element until you save it with the **save** command. When creating a new file, you must include a device alias with the **open** command. The following commands open a new waveaudio device element, start and stop recording, save the file element, and close the device element:

```
open new type waveaudio alias capture
record capture
stop capture
save capture orca.wav
close capture
```

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 string interface lets you use several shortcuts when working with MCI devices.

Using All as a Device Name

You can specify **all** as a *device_name* for any command that does not return information. When you specify **all**, the command is sent to all devices opened by your application. For example, "close all" closes all open devices and "play all" starts playing all devices opened by the task. Because MCI sends the commands to each device, there is a delay between when the first device receives the command and when the last device receives the command.

Combining the Device Type and Device Element Name

You can eliminate the **type** flag in the **open** command if you combine the device type with the device element name. MCI recognizes this combination when you use the following syntax:

device_type!*element_name*

The exclamation mark separates the device type from the element name. The following example opens the right.wav element with the waveaudio device:

```
open waveaudio!right.wav
```

Automatic Open

If MCI cannot identify the *device_name* as an already open device, MCI tries to automatically open the specified device. Automatic open does not let your application specify the **type** flag. If the device type is not supplied, MCI determines the device type from the element (filename) extensions listed in the system setup file. If you want to use a specific device, you can combine the device type name with the device element name using the exclamation mark.

Only the command-string interface supports automatic open. Automatic open will fail for device-specific commands. For example, a command to unlock the front panel of a videodisc player will fail an automatic open because this capability is specific to the particular videodisc player.

A device that was opened using the automatic open feature will not respond to a command that uses **all** as a device name.

Automatic Close

MCI automatically closes any device automatically opened using the command-string interface. MCI closes a device when the command completes, when you abort the command, when you request notification with a subsequent command, or when MCI detects a failure.

Using Wait and Notify Flags

Normally, MCI commands return to the user immediately, even if it takes several minutes to complete the action initiated by the command. For example, after a VCR device receives a rewind command, it returns before the tape has finished rewinding. You can use either of the following required MCI flags to modify this default behavior:

Flag	Description
notify	Directs the device to send an MM_MCINOTIFY message to a window when the requested action is complete.
wait	Directs the device to wait until the requested action is complete before returning to the application.

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. While the results of a notification are application-dependent, the application's window procedure can act upon four possible conditions associated with the notify message:

- Notification will occur when the notification conditions are satisfied.
- Notification can be superseded.
- Notification can be aborted.
- Notification can fail.

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” 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 normally.

A notification fails if a device error occurs while a device is executing the MCI command. For example, MCI posts this message when a hardware error occurs during a play command.

Obtaining Information From MCI Devices

Every device responds to the **capability**, **status**, and **info** commands. These commands obtain information about the device. For example, your application can determine if a videodisc requires a device element using the following command:

```
capability videodisc compound file
```

For most videodisc devices, this example would return **false**. The flags listed for the required and basic commands provide a minimum amount of information about a device. Many devices supplement the required and basic flags with extended flags to provide additional information about the device.

When you request information with the **capability**, **status**, or **info** command, the argument list can contain only one flag requesting information. The string interface can only return one string or value in response to a **capability**, **status**, or **info** command.

The Play Command

The **play** command starts playing a device. Without any flags, the **play** command starts playing from the current position and plays until the command is halted or until the end of the media or file is reached. For example, "play cdaudio" starts playing an audio disc from the position where it was stopped.

Most devices support the **play** command also support the **from** and **to** flags. These flags indicate the position at which the device should start and stop playing. For example, "play cdaudio from 0" plays the audio disc from the beginning of the first track. The units assigned to the position value depend on the device. For example, the position is normally specified in frames for CAV videodiscs, and milliseconds for digital audio.

As an extended command, devices add flags to use the capabilities of a particular device. For example, the **play** command for videodisc players adds the flags **fast**, **slow**, **reverse,** and **scan**.

Stop, Pause, and Resume Commands

The **stop** command suspends the playing or recording of a device. Many devices include the basic command **pause**, which also suspends these sessions. 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. If you want to continue playing or recording but want to stop at a position previously specified, use the **to** flag with these commands and repeat the position value.

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.

MCI System Commands

The following commands are interpreted directly by MCI. The remaining command tables list commands interpreted by the devices.

Command	Description	
sound	The device name of this command specifies a sound defined in a system setup file.. If it is not found, MCI uses a system default sound.	
sysinfo *item*	Obtains MCI system information. One of the following *items* modifies **sysinfo**:	
	installname	Returns the name used to install the device.
	quantity	Returns the number of MCI devices of the type specified by the device-name field. The device-name field must contain a standard MCI device type. Any digits after the name are ignored. The special device name **all** returns the total number of MCI devices in the system.
	quantity open	Returns the number of open MCI devices of the type specified by the device name. The device name must be a standard MCI device type. Any digits after the name are ignored. The special device name **all** returns the total number of MCI devices in the system that are open.
	name *index*	Returns the name of an MCI device. The *index* ranges from 1 to the number of devices of that type. If **all** is specified for the device name, *index* ranges from 1 to the total number of devices in the system.
	name *index* **open**	Returns the name of an open MCI device. The *index* ranges from 1 to the number of devices of that type. If **all** is specified for the device name, *index* ranges from 1 to the total number of devices in the system.

Required Commands for All Devices

The following commands are recognized by all devices. Extended commands can add other options to these commands. A list of the errors common to all the commands follows the required command table.

Command	Description	
capability *item*	Requests information about a particular capability of a device. While other capabilities are defined for specific devices and device types, the following *items* are always available:	
	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 requires an element name.
	device type	Returns one of the following: **audio tape** **cdaudio** **digital audio tape** **scanner** **sequencer** **videodisc** **videotape** **waveaudio**
	has audio	Returns **true** if the device supports audio playback.
	has video	Returns **true** if the device supports video.
	uses files	Returns **true** if the element of a compound device is a file pathname.
close	When sent to a simple device, closes the device. When sent to a compound device element, closes the element and any resources associated with it. MCI unloads a device when it is no longer being used.	
info *item*	Fills a user-supplied buffer with a NULL-terminated string containing textual information. One of the following *item* modifies **info**:	
	product	Returns a description of the hardware associated with a device. This usually includes the manufacturer and model information.
open *items*	Initializes the device. The following optional *items* modify **open**:	
	alias *device alias*	Specifies an alternate name for the given device. If specified, it must be used for subsequent references.
	shareable	Initializes the device or 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 it is already open and not shareable.
	type *device type*	Specifies the compound device that controls a device element. As an alternative to **type**, MCI can use the filename extension entries to select the device based on the extension used by the device element.
status *item*	Obtains status information for the device. One of the following *items* modifies **status**:	
	mode	Returns the current mode of the device. Commonly supported standard modes are: **not ready**, **paused**, **playing**, **stopped**, **open**, **recording**, and **seeking**
	ready	Returns **true** if the device is ready.

Basic Commands for Specific Device Types

In addition to the commands described previously, each device supports a set of commands specific to its device type. Where possible, these type-specific commands are identical between types. When type-specific commands are common to multiple devices, they are considered basic commands. For example, the basic **play** command is identical for videodisc and videotape players. Other basic commands are listed in the following table. Although these commands are optional for a device, if a command is used it must recognize all options listed in this table. The options generally provide for a minimum set of capabilities, but some devices may return "unsupported function" if an option is used which clearly doesn't apply.

Command	Description	
load *item*	Load a device element from disk. The following optional *item* modifies **load**:	
	filename	Specifies the source path and file.
pause	Pauses playing or recording.	
play *items*	Start playing the device. The following optional *items* modify **play**:	
	from *position* **to** *position*	Specifies the position to start and stop playing. If **from** is omitted, the play starts from the current position; if **to** is omitted, the play stops at the end of the media.

record *items*	Start recording data. All data recorded after a file is opened is discarded if the file is closed without saving it. The following optional *items* modify **record**:	
	insert	Specifies that new data is added to the device element at the current position.
	from *position* **to** *position*	Specifies the positions to start and stop recording. If **from** is omitted, the device starts recording at the current position; if **to** is omitted, the device records until a **stop** or **pause** command is received.
	overwrite	Specifies that new data will replace data in the device element.
	The default recording mode (**insert** or **overwrite**) depends on the specific device. Each device should define a default recording mode.	
resume	Resumes playing or recording following a pause.	
save *item*	Saves the MCI element. The following optional *item* modifies **save**:	
	filename	Specifies the destination path and file.
seek *item*	Moves to the specified position and stops. One of the following is required for *item* :	
	to *position*	Specifies the position to stop the seek.
	to start	Seeks to the start of the media or device element.
	to end	Seeks to the end of the media or device element.
set *items*	Sets the various control *items*:	
	audio all off **audio all on**	Enables or disables audio output
	audio left off **audio left on**	Enables or disables output to the left audio channel.
	audio right off **audio right on**	Enables or disables output to the right audio channel.
	door closed	Loads the media and closes the door if possible.
	door open	Opens the door and ejects the tray if possible.
	time format milliseconds	Sets time format to milliseconds. All position information is this format after this command. You can abbreviate milliseconds as **ms**.
	video off **video on**	Enables or disables video output.
status *item*	Obtains status information for the device. One of the following *items* modifies **status**:	
	current track	Returns the current track.

	length	Returns the total length of the segment.
	length track *track_number*	Returns the length of the serial 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*.
	start position	Returns the starting position of the media or device element.
	time format	Returns the time format.
stop	Stops the device.	

CD Audio (Redbook) Commands

The CD audio command set provides a common method for playing CD audio sequencesCD audio devices support the following core set of commands:

Command	Description	
capability *item*	Requests information about the capabilities of the CD audio device. One of the following *items* is required:	
	can eject	Returns **true** if the CD audio device can eject the media.
	can play	Returns **true** if the CD audio device can play the media.
	can record	Returns **false**.
	can save	Returns **false**.
	compound device	Returns **false**.
	device type	Returns **CDaudio**.
	has audio	Returns **true**.
	has video	Returns **false**.
	uses files	Returns **false**..
close	Closes the device.	
info *item*	Fills a user-supplied buffer with a NULL-terminated string containing textual information. One of the following optional *item* modifies **info**:	
	product	Returns the product name and model of the current audio device.
open *items*	Initializes the device. MCI reserves cdaudio for the compact disc audio device type. The following optional *items* modify **open**:	
	alias *device_ alias*	Specifies an alternate name for the given device. If specified, it must also be used for subsequent references.
	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 error if it is already open and not shareable.
pause	Pauses playing.	
play *items*	Starts playing audio. The following optional *items* modify **play**:	
	from *position* to *position*	Specifies the position to start and stop playing.
resume	Resumes playing from a paused state.	
seek *item*	Moves to the specified location on the disc. If already playing or recording, the device is stopped. One of the following *items* modifies **seek**:	

	to *position*	Specifies the destination position for the seek. If it is greater than the length of the disc, an out-of-range error is returned.
	to start	Specifies to seek to the start of the audio data on the CD.
	to end	Specifies to seek to the end of the audio data on the CD.
set *items*	Sets the various control *items*:	
	audio all off **audio all on**	Enables or disables audio output..
	audio left off **audio left on**	Enables or disables output to the left audio channel.
	audio right off **audio right on**	Enables or disables 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 position information is this format after this command. You can abbreviate milliseconds as **ms**.
	time format msf	Sets the time format to *mm*:*ss*:*ff*, where *mm* is minutes, *ss* is seconds, and *ff* is frames. All position information is in this format after this command. On input, *ff* can be omitted if 0, and *ss* can be omitted if both it and *ff* are 0. These fields have the following maximum values: Minutes 99 Seconds 59 Frames 74
	time format tmsf	Sets the time format to tt:mm:ss:ff where "tt" is tracks, "mm" is minutes, "ss" is seconds, and "ff" is frames. All position information is in this format after this command. On input "ff" can be omitted if 0, "ss" can be omitted if both it and "ff" are 0, and "mm" can be omitted if it, "ss" and "ff" are 0. These fields have the following maximum values: Tracks 99 Minutes 99 Seconds 59 Frames 74
status *item*	Obtains status information for the device. One of the following *items* modifies **status**:	
	current track	Returns the current track.
	length	Returns the total length of the disc.

	length track *track_number*	Returns the length of the track specified by track_number.
	media present	Returns **true** if the CD is inserted in the drive; otherwise, it returns **false**.
	mode	Returns **not ready**, **open**, **paused**, **playing**, **seeking**, or **stopped** for the current mode of the drive.
	number of tracks	Returns the number of tracks on the CD.
	position	Returns the current position.
	position track *track_no*	Returns the starting position of the track specified by *track_no*.
	ready	Returns **true** if the drive is ready.
	start position	Returns the starting position of the CD.
	time format	Returns the current time format.
stop	Stops playing.	

MIDI Sequencer Commands

The MIDI sequencer supports the following set of commands:

Command	Description	
capability *item*	Requests additional information about the capabilities of the MIDI sequencer. One of the following *items* is required:	
	can eject	Returns **false**..
	can play	Returns **true** if the sequencer can play.
	can record	Returns **true** if the sequencer can record MIDI data.
	can save	Returns **true** if the sequencer can save MIDI data.
	compound device	Generally returns **true**; most sequencers are compound devices..
	device type	Returns **sequencer**.
	has audio	Returns **true**.
	has video	Returns **false**.
	uses files	Returns **true**.
close	Closes the sequencer element and the port and file associated with it.	
info *item*	Fills a user-supplied buffer with a NULL-terminated string containing textual information. One of the following optional *item* modifies **info**:	
	product	Returns the product name of the current MIDI sequencer.
open *items*	Initializes the sequencer. The following optional *items* modify **open**:	
	alias *device_ alias*	Specifies an alternate name for the sequencer element. If specified, it must also be used for subsequent references.
	shareable	Initializes the sequencer element 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 it is already open and not shareable.
	type *device_ type*	MCI reserves sequencer for the MIDI sequencer device type. As an alternative to **type**, MCI can use the element filename extension entries to select the sequencer.
pause	Pauses playing.	
play *items*	Starts playing the sequencer. The following optional *items* modify **play**:	
	from *position* **to** *position*	Specifies the positions to start and stop playing. If **from** is omitted, play starts at the

		current position; if **to** is omitted, play stops at the end of the file.
record *items*	Starts recording MIDI data. All data recorded after a file is opened is discarded if the file is closed without saving it. The following optional *items* modify **record**:	
	insert	Specifies that new data is added to the device element.
	from *position* **to** *position*	Specifies the positions to start and stop recording. If **from** is omitted, the device starts recording at the current position; if **to** is omitted, the device records until a **stop** or **pause** command is received.
	overwrite	Specifies that new data will replace data in the device element.
resume	Resumes playing or recording following a pause.	
save *item*	Saves the MCI element. The following item *modifies* **save**:	
	filename	The *filename* specifies the destination path and file.
seek *item*	Moves to the specified position in the file. One of the following *items* is required:	
	to *position*	Specifies the final position for the seek.
	to start	Specifies to seek to the start of the sequence.
	to end	Specifies to seek to the end of the sequence.
set *items*	Sets the various control *items*:	
	audio all off **audio all on**	Enables or disables audio output..
	audio left off **audio left on**	Enables or disables output to the left audio channel.
	audio right off **audio right on**	Enables or disables output to the right audio channel.
	master MIDI	Sets the MIDI sequencer as the synchronization source. Synchronization data is sent in MIDI format.
	master none	Inhibits the sequencer from sending synchronization data.
	master SMPTE	Sets the MIDI sequencer as the synchronization source. Synchronization data is sent in SMPTE format.
	offset *time*	Sets the SMPTE offset *time* in colon form (*hours*:*minutes*:*seconds*:*frames*). The offset is the beginning time of a SMPTE based sequence.
	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..
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.
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.
tempo *tempo_value*	Sets the tempo of the sequence according to the current time format. For a ppqn-based file, the *integer* is interpreted as beats per minute. For a SMPTE-based file, the *integer* is interpreted as frames per second.
time format milliseconds	Sets time format to milliseconds. All position information is specified as milliseconds following this command. The sequence file sets the default format to ppqn or SMPTE. You can abbreviate milliseconds as **ms**.
time format song pointer	Sets time format to song pointer (sixteenth notes). This can only be performed for a sequence of division type ppqn.
time format SMPTE 24	Sets time format to SMPTE 24 frame rate. All position information is specified in SMPTE format following this command. The sequence file sets the default format to ppqn or SMPTE.
time format SMPTE 25	Sets time format to SMPTE 25 frame rate. All position information is specified in SMPTE format following this command. The sequence file sets the default format to ppqn or SMPTE.
time format SMPTE 30	Sets time format to SMPTE 30 frame rate. All position information is specified in SMPTE format following this command. The sequence file sets the default format to ppqn or SMPTE.
time format SMPTE 30 drop	Sets time format to SMPTE 30 drop frame rate. All position information is specified in SMPTE format following this command. The sequence file sets the default format to ppqn or SMPTE.

status *item* — Obtains status information for the MIDI sequencer. One of the following *items* modifies **status**:

current track	Returns the current track number.
division type	Returns one of the following file division type: PPQN, SMPTE 24 frame, SMPTE 25 frame, SMPTE 30 drop frame, or SMPTE 30 frame. Use this information to determine the format of the MIDI file, and the meaning of tempo and position information.
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 in colon form (*hours*:*minutes*:*seconds*:*frames*).
length track *track_number*	Returns the length of a sequence using the current time format. For ppqn files, this will be song pointer units. For SMPTE files, this will be in colon form (*hours*:*minutes*:*seconds*:*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.
offset	Returns the offset of a SMPTE-based file. The time is returned in colon form (*hours*:*minutes*:*seconds*:*frames*). The offset is the starting time of a SMPTE based sequence.
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 colon form (*hours*:*minutes*:*seconds*:*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 colon form (*hours*:*minutes*:*seconds*:*frames*).
ready	Returns **true** if the device is ready.
slave	Returns **file**, **midi**, **none**, or **smpte** depending on the type of synchronization set.
start position	Returns the starting position of the media or device element.
tempo	Returns the current tempo of a sequence in the current time format. For files with ppqn format, the tempo is in beats per minute. For files with SMPTE format, the tempo is in frames per second.

	time format	Returns the time format.
stop	Stops playing.	

Videodisc Player Commands

Videodisc players support the following core set of commands:

Command	Description	
capability *item*	Reports the capabilities of the device. The device should report capabilities according to the type of disc (CAV or CLV) inserted in the drive. If no disc is inserted, the device should assume CAV. One of the following optional *items* modifies **capability**:	
	can eject	Returns **true** if the device can eject the media.
	can play	Returns **true** if the device supports playing.
	can record	Returns **true** if the video device can record.
	can reverse	Returns **true** if the device can play in reverse, **false** otherwise. This is always **false** if a CLV disc is inserted.
	can save	Returns **false**.
	compound device	Returns **false**.
	device type	Returns **videodisc**.
	fast play rate	Returns the standard fast play rate of the player in frames per second. Returns 0 if the device cannot play fast.
	has audio	Returns **true** if the videodisc player has audio.
	has video	Returns **true**.
	media type	Returns **CAV**, **CLV**, or **other**, depending on the type of videodisc.
	normal play rate	Returns the normal play rate in frames per second. Returns 0 for CLV discs.
	slow play rate	Returns the standard slow play rate in frames per second. Returns 0 if the device cannot play slow.
	uses files	Returns **false**.
close	Closes the device.	
escape *item*	Sends custom information to a device. The following *item* modifies **escape**:	
	string	Specifies the custom infomation sent to the device.
info *item*	Fills a user-supplied buffer with a NULL-terminated string containing textual information. The following optional *item* modifies **info**:	
	product	Returns the product name of the device that the peripheral is controlling**..**
open *items*	Initializes the device. MCI reserves video disc for the videodisc device type. The following optional *items* modify **open**:	

	alias *device_ alias*	Specifies an alternate name for the given device. If specified, it must also be used for subsequent references.
	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 it is already open and not shareable.
pause	Stops playing. If a CAV disc is playing, it also freezes the video frame. If a CLV disc is playing, the player is stopped.	
play *items*	Starts playing. The following optional *items* modify **play:**	
	fast **slow**	Indicates that the device should play faster or slower than normal. To determine the exact speed on a particular player, use the **status speed** command. To specify the speed more precisely, use the **fps** flag. **Slow** applies only to CAV discs.
	from *position* **to** *position*	Specifies the positions to start and stop playing. Positions are in frames for CAV discs and in seconds for CLV discs, unless **chapter** is also used (in which case, the position is given in chapters). If **from** is omitted, play starts at the current position; if **to** is omitted, the play stops at the end of the disc.
	reverse	Sets the play direction to backwards. This applies only to CAV discs.
	scan	Indicates the play speed is as fast as possible, possibly with audio disabled. This applies only to CAV discs.
	speed *integer*	Specifies the rate of play. Currently supported speed values are measured in frames per second, which is the default. This applies only to CAV discs.
resume	Resumes playing.	
seek *item*	Searches using fast forward or fast reverse with video and audio off. The following optional *items* modify **seek**:	
	reverse	Indicates the seek direction on CAV discs is backwards. This modifier is invalid if **to** is specified.
	to *position*	Specifies the end position to stop the seek. If **to** is not specified, the seek continues until the end of the media is reached.
	to start	Specifies to seek to the start of the disc.
	to end	Specifies to seek to the end of the disc.
set *items*	Sets the various control *items*:	
	audio all off **audio all on**	Enables or disables audio output.

	audio left off **audio left on**	Enables or disables output to the left audio channel.
	audio right off **audio right on**	Enables or disables output to the right audio channel.
	door open	Opens the door and ejects the tray, if possible.
	door closed	Retracts the tray and closes the door, if possible.
	time format frames	Sets the position format to frames on CAV discs. All position information is specified in this format following this command. This is the default for CAV discs.
	time format hms	Sets position format to *h*:*mm*:*ss* where *h* is hours, *mm* is minutes, and *ss* is seconds. All position information is specified in this format following this command. On input, *h* may be omitted if 0, and *mm* may be omitted if both it and *h* are 0. This is the default for CLV discs.
	time format milliseconds	Sets the position format to milliseconds. All position information is specified in this format following this command. You can abbreviate milliseconds as **ms**.
	time format track	Sets the position format to tracks (chapters). All position information is specified in this format following this command.
	video on **video off**	Turns the video on or off.
spin *item*	Starts the disc spinning or stops the disc from spinning. One of the following *items* modifies **status**:	
	down	Stops the disc from spinning.
	up	Starts the disc spinning.
status *item*	Obtains status information for the device. One of the following *items* modifies **status**:	
	current track	Returns the current track (chapter) number.
	disc size	Returns either 8 or 12 to indicate the size of the loaded disc in inches.
	forward	Returns **true** if the play direction is forward or if the device is not playing; **false** if the play direction is backward.
	length	Returns the total length of the segment.
	length track *track_number*	Returns the length of the track (chapter) specified by *track_number*.
	media present	Returns **true** if a disc is inserted in the device, **false** otherwise.
	media type	Returns either **CAV**, **CLV**, or **other** depending on the type of videodisc.
	mode	Returns **not ready**, **opened**, **paused**, **parked**,

		playing, **scanning**, **seeking**, or **stopped**.
	number of tracks	Returns the number of tracks (chapters) on the media.
	position	Returns the current position.
	position track *track_number*	Returns the position of the start of the track (chapter) specified by *track_number*.
	ready	Returns **true** if the device is ready.
	side	Returns 1 or 2 to indicate which side of the disc is loaded.
	speed	Returns the speed in frames per second.
	start position	Returns the starting position of the disc.
	time format	Returns the time format.
step *items*	Step the play one or more frames forward or backward. The default action is to step one frame forward. The **step** command applies only to CAV discs. The following *items* modifies **step**:	
	by *frames*	Specifies the number of *frames* to step. If a negative value is used, the **reverse** flag is ignored.
	reverse	Step backward.
stop	Stop playing.	

Waveform Audio Commands

Waveform audio drivers must support the following core set of commands:

Command	Description	
capability *item*	Requests additional information about the capabilities of the waveform audio driver. One of the following *items* modify **capability**:	
	can eject	Returns **false**.
	can play	Returns **true** if the device can play. The wave audio device returns true if an output device is available.
	can record	Returns **true** if the waveform driver can record. The waveform audio device returns **true** if an input device is available.
	can save	Returns **true** if the wave audio device can save data.
	compound device	Generally returns **true**; most waveform audio devices are compound devices.
	device type	Returns **waveaudio**.
	has audio	Returns **true**
	has video	Returns **false**.
	inputs	Returns the total number of input devices.
	outputs	Returns the total number of output devices.
	uses files	Returns **true**.
close	Closes the device element and any resources associated with it.	
cue *item*	Prepares for playing or recording. The **cue** command does not have to be issued prior to playing or recording. However, depending on the device, it might reduce the delay associated with the **play** or **record** command. This command fails if playing or recording is in progress. The *item* is one of the following:	
	input	Prepares for recording.
	output	Prepares for playing. This is the default.
delete *items*	Deletes a data segment from the MCI element. The following optional *items* modify **delete**:	
	from *position* **to** *position*	Specifies the positions to start and stop deleting data. If **from** is omitted, the deletion starts at the current position; if **to** is omitted, the deletion stops at the end of the file or waveform.
info *item*	Fills a user-supplied buffer with a NULL-terminated string containing textual information. One of the following *items* modifies **info**:	
	file	Returns the current filename.

	product	Returns the product name of the current audio output device.
	input	Returns the product name of the current waveform input device or **none** if no device is set.
	output	Returns the product name of the current waveform output device or **none** if no device is set.
open *items*	Initializes the device. The following *items* are optional:	
	alias *device_ alias*	Specifies an alternate name for the given device. If specified, it must also be used the alias for references.
	buffer *buffer_size*	Sets the size in seconds of the buffer used by the wave audio device. The default size of the buffer is set when the wave audio device is installed or configured. Typically, the buffer size is set to 4 seconds.
	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 it is already open and not shareable.
	type *device_type*	Specifies the compound device used to control a device element. MCI reserves waveaudio for the waveform audio device type. As an alternative to **type**, MCI can use the element filename extension entries to select the controlling device
pause	Pauses playing or recording.	
play *items*	Starts playing audio. The following optional *items* modify **play**:	
	from *position* **to** *position*	Specifies the positions to start and stop playing. If **from** is omitted, play starts at the current position; if **to** is omitted, play stops at the end of the file or waveform.
record *items*	Starts recording audio. All data recorded after a file is opened is discarded if the file is closed without saving it. The following optional *items* modify **record**:	
	insert	Specifies that new data is added to the device element.
	from *position* **to** *position*	Specifies the positions to start and stop recording. If **from** is omitted, the device starts recording at the current position; if **to** is omitted, the device records until a **stop** or **pause** command is received.
	overwrite	Specifies that new data will replace data in the device element.
resume	Resumes playing or recording following a pause.	
save *item*	Saves the MCI element in its current format. The following item	

modifies **save**:

filename	Specifies the file and pathname used to save data.

seek *item*

Moves to the specified location in the file. Playback or recording is stopped after the seek. One of the following items modify **seek**:

to *position*	Specifies the stop position.
to start	Specifies to seek to the first sample.
to end	Specifies to seek to the last sample.

set *items*

Sets the following control *items*:

alignment *integer*	Sets the alignment of data blocks. The file is saved in the new format.
any input	Use any input that supports the current format when recording. This is the default.
any output	Use any output that supports the current format when playing. This is the default.
audio all off **audio all on**	Enables or disables audio output.
audio left off **audio left on**	Enables or disables output to the left audio channel.
audio right off **audio right on**	Enables or disables output to the right audio channel.
bitspersample *bit_count*	Sets the number of bits per sample played or recorded. The file is saved in this format.
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 channel count 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	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 position information is specified as milliseconds following this command. 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.	
status *item*	Obtains status information for the device. One of the following *items* modifies **status**:	
	alignment	Returns the block alignment of data in bytes.
	bitspersample	Returns the bits per sample.
	bytespersec	Returns the average number of bytes per second played or recorded.
	channels	Returns the number of channels set (1 for mono, 2 for stereo).
	current track	Returns the index of the current track.
	format tag	Returns the format tag.
	input	Returns the currently set input. If no input is set, the error returned indicates that any device can be used.
	length	Returns the total length of the waveform.
	length track *track_number*	Returns the length of the waveform track.
	level	Returns the current audio sample value.
	media present	Returns **true**.
	mode	Returns **not ready, paused**, **playing**, **stopped**, **recording**, or **seeking**.
	number of tracks	Returns the number of tracks (chapters).
	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*.
	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 waveform data.
	time format	Returns the current time format.
stop	Stops playing or recording.	