Microsoft® Windows™ 3.1

Programmer's Reference

Volume 2

Functions

Microsoft® Windows™ 3.1

Programmer's Reference

Volume 2

Functions

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Microsoft Windows programmer's reference.
p. cm.
Includes indexes.
Contents: v. 1. Overview -- v. 2. Functions -- v. 3. Messages, structures, macros -- v. 4. Resources.
ISBN 1-55615-453-4 (v. 1). -- ISBN 1-55615-463-1 (v. 2). -- ISBN 1-55615-464-X (v. 3). -- ISBN 1-55615-494-1 (v. 4)
1. Microsoft Windows (Computer program) I. Microsoft Corporation.
QA76.76.W56M532 1992
005.4'3--dc20 91-34199
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 7 6 5 4 3 2

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Document No. PC28916-0492

Introduction

The Microsoft® Windows™ 3.1 operating system is a single-user system for personal computers. Applications that run with this operating system use functions in the Windows applications programming interface (API). This manual describes the API functions in alphabetic order, including each function's purpose, the version of Windows in which it first appeared, and the function's syntax, parameters, and possible return values. Many function descriptions also contain additional information and simple code examples that illustrate how the function can be used to carry out simple tasks.

How to Use This Manual

For most of the functions described in this manual, the syntax is given in C-language format. In your C-language source files, the function name must be spelled exactly as given in syntax and the parameters must be used in the order given in syntax.

The Windows API uses many types, structures, and constants that are not part of standard C language. These items, designed for Windows, are defined in the Windows C-language header files. Although there are many Windows header files, the majority of API functions, structures, and messages are defined in the WINDOWS.H header file. You can use these items in your Windows application by placing an **#include** directive specifying WINDOWS.H at the beginning of your C-language source file.

In this manual, if a function is not defined in WINDOWS.H, its appropriate header file is included in the first line of syntax. If no header file is listed, you can assume the function is defined in WINDOWS.H.

Note You will find a list of the appropriate module and library for each Windows function in the *Microsoft Windows Programmer's Reference, Volume 1*. A list of the types used in the Windows API, with a brief description of each, is provided in the *Microsoft Windows Programmer's Reference, Volume 3*.

Document Conventions

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

Convention	Meaning
Bold text	Denotes a term or character to be typed literally, such as a resource-definition statement or function name (**MENU** or **CreateWindow**), a Microsoft MS-DOS® command, or a command-line option (**/nod**). You must type these terms exactly as shown.
Italic text	Denotes a placeholder or variable: You must provide the actual value. For example, the statement **SetCursorPos**(*X*, *Y*) requires you to substitute values for the *X* and *Y* parameters.
[ ]	Enclose optional parameters.
|	Separates an either/or choice.
...	Specifies that the preceding item may be repeated.

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

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

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

Contents

AbortDoc

3.1

int AbortDoc(*hdc*)
HDC *hdc*; /* handle of device context */

The **AbortDoc** function terminates the current print job and erases everything drawn since the last call to the **StartDoc** function. This function replaces the ABORTDOC printer escape for Windows version 3.1.

Parameters

hdc
Identifies the device context for the print job.

Return Value

The return value is greater than or equal to zero if the function is successful. Otherwise, it is less than zero.

Comments

Applications should call the **AbortDoc** function to terminate a print job because of an error or if the user chooses to cancel the job. To end a successful print job, an application should use the **EndDoc** function.

If Print Manager was used to start the print job, calling the **AbortDoc** function erases the entire spool job—the printer receives nothing. If Print Manager was not used to start the print job, the data may have been sent to the printer before **AbortDoc** was called. In this case, the printer driver would have reset the printer (when possible) and closed the print job.

See Also

EndDoc, **SetAbortProc**, **StartDoc**

AbortProc

3.1

BOOL CALLBACK AbortProc(*hdc*, *error*)
HDC *hdc*; /* handle of device context */
int *error*; /* error value */

The **AbortProc** function is an application-defined callback function that is called when a print job is to be canceled during spooling.

Parameters

hdc
Identifies the device context.

error
Specifies whether an error has occurred. This parameter is zero if no error has occurred; it is SP_OUTOFDISK if Print Manager is currently out of disk space

and more disk space will become available if the application waits. If this parameter is SP_OUTOFDISK, the application need not cancel the print job. If it does not cancel the job, it must yield to Print Manager by calling the **PeekMessage** or **GetMessage** function.

Return Value

The callback function should return TRUE to continue the print job or FALSE to cancel the print job.

Comments

An application installs this callback function by calling the **SetAbortProc** function. **AbortProc** is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also

GetMessage, PeekMessage, SetAbortProc

AccessResource

2.x

int AccessResource(*hinst***, *hrsrc*)**
HINSTANCE *hinst*; /* handle of module with resource */
HRSRC *hrsrc*; /* handle of resource */

The **AccessResource** function opens the given executable file and moves the file pointer to the beginning of the given resource.

Parameters

hinst
Identifies the instance of the module whose executable file contains the resource.

hrsrc
Identifies the desired resource. This handle should be created by using the **FindResource** function.

Return Value

The return value is the handle of the resource file if the function is successful. Otherwise, it is –1.

Comments

The **AccessResource** function supplies an MS-DOS file handle that can be used in subsequent file-read calls to load the resource. The file is opened for reading only.

Applications that use this function must close the resource file by calling the **_lclose** function after reading the resource. **AccessResource** can exhaust available MS-DOS file handles and cause errors if the opened file is not closed after the resource is accessed.

In general, the **LoadResource** and **LockResource** functions are preferred. These functions will access the resource more quickly if several resources are being read, because Windows maintains a file-handle cache for accessing executable files. However, each call to **AccessResource** requires that a new handle be opened to the executable file.

You should not use **AccessResource** to access executable files that are installed in ROM on a ROM-based system, since there are no disk files associated with the executable file; in such a case, a file handle cannot be returned.

See Also **FindResource, _lclose, LoadResource, LockResource**

AddAtom

2.x

ATOM AddAtom(*lpszName*)
LPCSTR *lpszName*; /* address of string to add */

The **AddAtom** function adds a character string to the local atom table and returns a unique value identifying the string.

Parameters

lpszName
Points to the null-terminated character string to be added to the table.

Return Value

The return value specifies the newly created atom if the function is successful. Otherwise, it is zero.

Comments

The **AddAtom** function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom value and increments (increases by one) the string's reference count.

The **MAKEINTATOM** macro can be used to convert a word value into a string that can be added to the atom table by using the **AddAtom** function.

The atom values returned by **AddAtom** are in the range 0xC000 through 0xFFFF.

Atoms are case-insensitive.

Example

The following example uses the **AddAtom** function to add the string "This is an atom" to the local atom table:

```
ATOM at;
char szMsg[80];

at = AddAtom("This is an atom");

if (at == 0)
    MessageBox(hwnd, "AddAtom failed", "", MB_ICONSTOP);
else {
    wsprintf(szMsg, "AddAtom returned %u", at);
    MessageBox(hwnd, szMsg, "", MB_OK);
}
```

See Also **DeleteAtom**, **FindAtom**, **GetAtomName**

AddFontResource

2.x

int AddFontResource(*lpszFilename*)
LPCSTR *lpszFilename*; /* address of filename */

The **AddFontResource** function adds a font resource to the Windows font table. Any application can then use the font.

Parameters

lpszFilename
Points to a character string that names the font resource file or that contains a handle of a loaded module. If this parameter points to a font resource filename, it must be a valid MS-DOS filename, including an extension, and the string must be null-terminated. The system passes this string to the **LoadLibrary** function if the font resource must be loaded.

Return Value

The return value specifies the number of fonts added if the function is successful. Otherwise, it is zero.

Comments

Any application that adds or removes fonts from the Windows font table should send a WM_FONTCHANGE message to all top-level windows in the system by using the **SendMessage** function with the *hwnd* parameter set to 0xFFFF.

When font resources added by using **AddFontResource** are no longer needed, you should remove them by using the **RemoveFontResource** function.

Example

The following example uses the **AddFontResource** function to add a font resource from a file, notifies other applications by using the **SendMessage** function, then removes the font resource by using the **RemoveFontResource** function:

```
AddFontResource("fontres.fon");
SendMessage(HWND BROADCAST, WM FONTCHANGE, 0, 0);
    .
    . /* Work with the font. */
    .
if (RemoveFontResource("fontres.fon")) {
    SendMessage(HWND BROADCAST, WM FONTCHANGE, 0, 0);
    return TRUE;
}
else
    return FALSE;
```

See Also **LoadLibrary, RemoveFontResource, SendMessage**

AdjustWindowRect

2.x

void AdjustWindowRect(*lprc***, *dwStyle*, *fMenu*)**
RECT FAR* *lprc*; /* address of client-rectangle structure */
DWORD *dwStyle*; /* window styles */
BOOL *fMenu*; /* menu-present flag */

The **AdjustWindowRect** function computes the required size of the window rectangle based on the desired client-rectangle size. The window rectangle can then be passed to the **CreateWindow** function to create a window whose client area is the desired size.

Parameters

lprc

Points to a **RECT** structure that contains the coordinates of the client rectangle. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
    int left;
    int top;
    int right;
    int bottom;
} RECT;
```

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

dwStyle

Specifies the window styles of the window whose client rectangle is to be converted.

fMenu

Specifies whether the window has a menu.

Return Value This function does not return a value.

Comments A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window.

AdjustWindowRect does not take titles and borders into account when computing the size of the client area. For window styles that include titles and borders, applications must add the title and border sizes after calling **AdjustWindowRect**. This function also does not take the extra rows into account when a menu bar wraps to two or more rows.

See Also **AdjustWindowRectEx**, **CreateWindowEx**

AdjustWindowRectEx

3.0

void AdjustWindowRectEx(*lprc*, *dwStyle*, *fMenu*, *dwExStyle*)

```
RECT FAR* lprc;      /* address of client-rectangle structure   */
DWORD dwStyle;       /* window styles                           */
BOOL fMenu;          /* menu-present flag                       */
DWORD dwExStyle;     /* extended style                          */
```

The **AdjustWindowRectEx** function computes the required size of the rectangle of a window with extended style based on the desired client-rectangle size. The window rectangle can then be passed to the **CreateWindowEx** function to create a window whose client area is the desired size.

Parameters

lprc

Points to a **RECT** structure that contains the coordinates of the client rectangle. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
    int left;
    int top;
    int right;
    int bottom;
} RECT;
```

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

dwStyle

Specifies the window styles of the window whose client rectangle is to be converted.

fMenu
Specifies whether the window has a menu.

dwExStyle
Specifies the extended style of the window being created.

Return Value

This function does not return a value.

Comments

A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window.

AdjustWindowRectEx does not take titles and borders into account when computing the size of the client area. For window styles that include titles and borders, applications must add the title and border sizes after calling **AdjustWindowRectEx**. This function also does not take the extra rows into account when a menu bar wraps to two or more rows.

See Also

AdjustWindowRect, CreateWindowEx

AllocDiskSpace

3.1

#include <stress.h>

int AllocDiskSpace(*lLeft*, *uDrive*)
long *lLeft*; /* number of bytes left available */
UINT *uDrive*; /* disk partition */

The **AllocDiskSpace** function creates a file that is large enough to ensure that the specified amount of space or less is available on the specified disk partition. The file, called STRESS.EAT, is created in the root directory of the disk partition.

If STRESS.EAT already exists when **AllocDiskSpace** is called, the function deletes it and creates a new one.

Parameters

lLeft
Specifies the number of bytes to leave available on the disk.

uDrive
Specifies the disk partition on which to create the STRESS.EAT file. This parameter must be one of the following values:

Value	Meaning
EDS_WIN	Creates the file on the Windows partition.
EDS_CUR	Creates the file on the current partition.
EDS_TEMP	Creates the file on the partition that contains the TEMP directory.

Return Value

The return value is greater than zero if the function is successful; it is zero if the function could not create a file; or it is –1 if at least one of the parameters is invalid.

Comments

In two situations, the amount of free space left on the disk may be less than the number of bytes specified in the *lLeft* parameter: when the amount of free space on the disk is less than the number in *lLeft* when an application calls **AllocDiskSpace**, or when the value of *lLeft* is not an exact multiple of the disk cluster size.

The **UnAllocDiskSpace** function deletes the file created by **AllocDiskSpace**.

See Also

UnAllocDiskSpace

AllocDStoCSAlias

3.0

UINT AllocDStoCSAlias(*uSelector*)
UINT *uSelector*; /* data-segment selector */

The **AllocDStoCSAlias** function accepts a data-segment selector and returns a code-segment selector that can be used to execute code in the data segment.

Parameters

uSelector
Specifies the data-segment selector.

Return Value

The return value is the code-segment selector corresponding to the data-segment selector if the function is successful. Otherwise, it is zero.

Comments

The application should not free the new selector by calling the **FreeSelector** function. Windows will free the selector when the application terminates.

In protected mode, attempting to execute code directly in a data segment will cause a general-protection violation. **AllocDStoCSAlias** allows an application to execute code that the application had created in its own stack segment.

Windows does not track segment movements. Consequently, the data segment must be fixed and nondiscardable; otherwise, the data segment might move, invalidating the code-segment selector.

The **PrestoChangoSelector** function provides another method of obtaining a code selector corresponding to a data selector.

An application should not use this function unless it is absolutely necessary, since its use violates preferred Windows programming practices.

See Also **FreeSelector, PrestoChangoSelector**

AllocFileHandles

3.1

#include <stress.h>

int AllocFileHandles(*Left***)**
int *Left*; /* number of file handles to leave available */

The **AllocFileHandles** function allocates file handles until only the specified number of file handles is available to the current instance of the application. If this or a smaller number of handles is available when an application calls **AllocFileHandles**, the function returns immediately.

Before allocating new handles, this function frees any handles previously allocates by **AllocFileHandles**.

Parameters

Left
Specifies the number of file handles to leave available.

Return Value

The return value is greater than zero if **AllocFileHandles** successfully allocates at least one file handle. The return value is zero if fewer than the specified number of file handles were available when the application called **AllocFileHandles**. The return value is −1 if the *Left* parameter is negative.

Comments

AllocFileHandles will not allocate more than 256 file handles, regardless of the number available to the application.

The **UnAllocFileHandles** function frees all file handles previously allocated by **AllocFileHandles**.

See Also **UnAllocFileHandles**

AllocGDIMem

3.1

#include <stress.h>

BOOL AllocGDIMem(*uLeft***)**
UINT *uLeft*; /* number of bytes to leave available */

The **AllocGDIMem** function allocates memory in the graphics device interface (GDI) heap until only the specified number of bytes is available. Before making any new memory allocations, this function frees memory previously allocated by **AllocGDIMem**.

Parameters

uLeft
Specifies the amount of memory, in bytes, to leave available in the GDI heap.

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero.

Comments

The **FreeAllGDIMem** function frees all memory allocated by **AllocGDIMem**.

See Also

FreeAllGDIMem

AllocMem

3.1

#include <stress.h>

BOOL AllocMem(*dwLeft***)**
DWORD *dwLeft*; /*smallest memory allocation */

The **AllocMem** function allocates global memory until only the specified number of bytes is available in the global heap. Before making any new memory allocations, this function frees memory previously allocated by **AllocMem**.

Parameters

dwLeft
Specifies the smallest size, in bytes, of memory allocations to make.

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero.

Comments

The **FreeAllMem** function frees all memory allocated by **AllocMem**.

See Also

FreeAllMem

AllocResource

2.x

HGLOBAL AllocResource(*hinst*, *hrsrc*, *cbResource*)
HINSTANCE *hinst*; /* handle of module containing resource */
HRSRC *hrsrc*; /* handle of resource */
DWORD *cbResource*; /* size to allocate, or zero */

The **AllocResource** function allocates uninitialized memory for the given resource.

Parameters

hinst
Identifies the instance of the module whose executable file contains the resource.

hrsrc
Identifies the desired resource. This handle should have been created by using the **FindResource** function.

cbResource
Specifies the size, in bytes, of the memory object to allocate for the resource. If this parameter is zero, Windows allocates enough memory for the specified resource.

Return Value

The return value is the handle of the global memory object if the function is successful.

See Also

FindResource, **LoadResource**

AllocSelector

3.0

UINT AllocSelector(*uSelector*)
UINT *uSelector*; /* selector to copy or zero */

The **AllocSelector** function allocates a new selector.

Do not use this function in an application unless it is absolutely necessary, since its use violates preferred Windows programming practices.

Parameters

uSelector
Specifies the selector to return. If this parameter specifies a valid selector, the function returns a new selector that is an exact copy of the one specified here. If this parameter is zero, the function returns a new, uninitialized sector.