

Grundlagen um APIs in Profan anwenden zu können.

Diese zusammengestellten Grundlagen sollen dabei helfen Windows APIs in Profan benutzen zu können. Das heißt, dass ein Anfänger in diesem Bereich, damit ein Basiswissen erwerben kann.

Profan ist zwar eine, wie ich meine, sehr gute und einfach zu benutzende Programmiersprache mit sehr großem Funktionsumfang. Doch können klarerweise nicht alle Möglichkeiten die Windows bietet inkludiert sein.

Windows bietet viele weitere Funktionen an. Diese Funktionen sind in verschiedenen DLLs im Windowsverzeichnis abgelegt.

Mit dem geeigneten Wissen kann man diese Funktionen auch in Profan nützen.

Diese Beschreibung bezieht sich auf die 32-Bit Windowsversion (ab Windows 95 aufwärts). Weiters muss die Profanversion (für diese Beschreibung) mindestens 7.5 sein (diese Version gibt es schon als Freeware!). Wieweit alle Funktionen auch noch unter Windows 7 (64-Bit) funktionieren kann ich derzeit nicht beurteilen.

Voraussetzung:

- eine Referenz muss vorhanden sein.

Als Referenz kann die Hilfedatei Win32.hlp, oder die 5 Bände „Das Win32API“, oder der Band „WIN32 Programmierung“ (API Bible) verwendet werden. Alle 3 Möglichkeiten haben individuelle Vor- und Nachteile. Wahrscheinlich gibt es aber noch andere Referenzhandbücher bzw. Hilfedateien.

Die Win32.hlp ist in englischer, die Bücher sind in deutscher Sprache.

Die Hilfedatei kann kostenlos vom Internet heruntergeladen werden und scheint die vollständigste Unterlage zu sein. Die Bücher kosten pro Band um die Euro 50,-. Inzwischen gibt es die Hilfedateien auch als .chm Version – leider auf mehrere Teile unterteilt.

Leider ist die Win32.hlp schon relativ veraltet und wird von Windows 7 gerade noch unterstützt (wie lange noch?). Ansonsten besteht die Möglichkeit über Google Informationen über API Funktionen zu erfahren (von der Microsoftseite direkt eher kaum). Aber wenn bei Google der Funktionsname als Suchargument (eventuell noch mit **msdn** ergänzt) eingegeben wird wird normalerweise gleich im ersten gefundenen Beitrag der *Suchbegriff + Function(Windows)* der richtige Beitrag sein. Hier ist die Parameterliste und, weiter unten unter Requirements in der Tabelle bei DLL, auch der Name des APIs angeführt.

Die hier erstellten Unterlagen sind sicher nicht vollständig, sie wurden aus meinen Erfahrungen (und Interpretationen der Beschreibungen) erstellt. Falls Fehler gefunden wurden bitte ich um Verständigung damit ich diese Unterlagen berichtigen kann.

Diese Grundlagen beschreiben nicht einzelne APIs sondern sie sollen nur, an 2 Beispielen, allgemein zeigen wie APIs verwendet werden können.

Ergänzend zur Hilfe Win32.hlp kann mein Programm API_Hilfe benutzt werden das die, zu den in der Win32.hlp angeführten symbolischen Namen, als hexadezimale (und dezimalen) Werte anzeigt.

Gerhard Putschalka 5.9.2002 neu überarbeitet 10.6.2010 gerhard-putschalka@gmx.at
Homepage <http://www.gerhard-putschalka.xprofan.com/index.htm>

Auf den H.P.s von Andreas Miethe (<http://www.ampsoft.eu>)
und
Uwe "Pascal" Niemeier (<http://www.tomcatsoft.de>)
Gibt es viele Beispiele zur Verwendung von APIs (und Controls).

[Literatur: ein Verzeichnis befindet sich im Anhang Seite 15](#)

Am Beispiel der WinApi-Funktion `GetTextExtentPoint32` möchte ich zeigen wie man vorgehen kann. Mit dieser Funktion kann das Rechteck festgestellt werden, das einen Text umschließt. Damit ist es möglich einen Text (z.B. mit `DrawText`) zentriert oder auch rechtsbündig zu positionieren.

Um ein API zu verwenden muss in Profan eine eigene Funktion definiert werden. Für unser Beispiel kann sie so aussehen:

```
Def GetWeite(4) ! "GDI32", "GetTextExtentPoint32A"
```

Wir beschreiben hier unsere Funktion und benennen sie `GetWeite_`. Der Name kann frei gewählt werden. Er darf nur nicht gleich dem Namen einer in Profan existierenden Funktion sein! Ob dieser Funktionsname mit einem `@` angeführt wird oder nicht hat keine Bedeutung (das Zeichen `@` musste in Profanversionen vor 4.0 geschrieben werden).

Die Anzahl der Parameter „(4)“ hängt mit der Funktion `GetTextExtentPoint32` zusammen. Warum sie jetzt plötzlich `GetTextExtentPoint32A` statt `GetTextExtentPoint32` heißt wird später erklärt. In der Win-Hilfe ist sie nur unter `GetTextExtentPoint32` zu finden).

Um die Begriffe „Funktion“ nicht durcheinander zu bringen: `GetWeite` ist der Funktionsname den wir in der Profansource verwenden. `GetTextExtentPoint32A` ist der Funktionsname im WinApi `GDI32`. Warum das WinApi `GDI32` heißt wird auch später erklärt.

Bleibt noch die Bedeutung des Rufzeichens. Dieses zeigt an, dass ein Win32API benutzt wird (wir arbeiten praktisch nur mit Win32APIs).

Sehen wir uns an, wie das API angewendet werden kann. In diesem Beispiel erfolgt die Berechnung der Textlänge in einer Prozedur mit dem Namen `TextWeite`.

Beim Aufruf der Prozedur wird im ersten Parameter der zu prüfende Text (als String) und im 2. Parameter ein Kennzeichen übergeben, das steuert ob die Berechnung für die Darstellung am Bildschirm (= 0) oder auf dem Drucker (= 1) erfolgen soll. Dieses Kennzeichen wird als Integer übergeben. Für den Drucker wird in der Prozedur der errechnete Wert durch 10 geteilt.

```
Proc TextWeite
Parameters T$, Prt_Ctl%
  Declare Leng&, Ber#, Lng#
  Dim Ber#, 255
  Dim Lng#, 8
  String Ber#, 0 = T$          ' setze den Text in die Bereichsvariable
  Leng& = @Len(T$)           ' und ermittle die Textlänge (Anzahl der Zeichen)

  ' rufe das API auf, übergib die Daten. Der 4. Parameter ist die Bereichsvariable
  ' für das Ergebnis: 2 Longints (Lng#,0 = Breite, Lng#,4 = Höhe)
  @GetWeite(%HDC, Ber#, Leng&, Lng#)
  Leng& = @Long(Lng#, 0)     ' übernahm nur die Breite
  Case (Prt_Ctl% = 1) : Leng& = (Leng& / 10)
  Dispose Ber#
  Dispose Lng#
  Return Leng&               ' der Retourwert der Prozedur ist die Textlänge in Pixel
EndProc
```

Daten (Strings und auch numerische Werte) werden normalerweise zwischen Profan und API mit Bereichsvariablen übergeben. Handelt es sich um einen numerischen Wert kann er auch mit einer `LongInt` Variablen (`Var&`) oder direkt als Zahl an die Funktion übergeben werden. Für Retourwerte werden nur Bereichsvariablen benutzt. Der Retourwert kann aber auch das Ergebnis der Funktion sein und auch eine `LongInt` Variable sein. Dies ist abhängig von der API-Funktion.

Eine Bereichsvariable zum Datenaustausch ist jedenfalls immer OK.

Wichtig dabei ist, dass sie immer ausreichend groß **DI**Mensioniert ist! Die Größe hängt von der jeweiligen API-Funktion ab. (zu groß kann aber nie schaden).

Diese Prozedur ist bewusst einfach gehalten. Für die API-Funktion werden 2 Bereichsvariable verwendet: `Ber#` und `Lng#`. `Ber#` dient als Eingabe des Textes zur Funktion und in `Lng#` gibt die API-Funktion die Textgröße (in Punkten) an Profan zurück.

Bevor weitere Erläuterungen zu dieser Prozedur folgen, wollen wir uns mit der Referenz befassen: warum und was müssen wir bei `@GetWeite` angeben (auch hier kann das Zeichen `@` entfallen)?

Sehen wir in der Hilfedatei `Win32.hlp` unter dem Stichwort `GetTextExtentPoint32` nach (ein `GetTextExtentPoint32A` werden wir hier nicht finden).



Bild 1 Angaben zu GetTextExtentPoint32 aus der Win32.hlp

Mit einem Click auf den Button „Quick Info“ erhalten wir das nächste Bild:

GetTextExtentPoint32	
Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT; Win95
Platform Notes	Windows 95: int == 16 bits

Bild 2 weitere Informationen über die Funktion GetTextExtentPoint32.

Hier sehen wir welches API die Funktion `GetTextExtentPoint32` enthält. Es wird hier nur der Name `GDI32` ohne der Dateierweiterung bei der Definition der Profanfunktion eingesetzt.

Bleibt noch zu klären warum `GetTextExtentPoint32A` statt `GetTextExtentPoint32` ?

Normalerweise wird in Texten jeweils ein Zeichen in einem Byte gespeichert. Es gibt aber auch den Unicode der 2 Bytes für ein Zeichen belegt (wie sonst könnte man bei den fernöstlichen Sprachen die vielen Zeichen verschlüsseln?). Daher gibt es viele API-Funktionen doppelt. Zur Unterscheidung wird für den 1-Byte Code an den Funktionsnamen ein **A** angehängt, für den Unicode wird ein **W** angehängt .

Ich denke, dass normalerweise nur der 1-Byte Code angewendet wird somit ist **A** bei uns richtig.

Im Bild 2 gibt es auch eine Zeile „Unicode“. Ich nehme an, wenn hier nicht **No** angegeben ist, so handelt es sich um die Doppelversion (**A** oder **W**) und es ist daher ein **A** an den Funktionsnamen anzuhängen.

Wenn also beim Aufruf der Funktion in Profan die Meldung kommt: „Funktion unbekannt“ so ist es wahrscheinlich, dass an den Funktionsnamen das **A** angehängt werden sollte – oder es dort nicht sein sollte.

Doch zurück zum Bild 1. Unter

```
BOOL GetTextExtentPoint32(  
HDC hdc,           // handle of device context  
LPCTSTR lpstring, // address of textstring  
int cbstring,     // number of characters in string  
LPSIZE lpsize     // address of structure for string size  
);
```

sehen wir (innerhalb der Klammern) 4 Zeilen. Das heißt es sind 4 Parameter an die API-Funktion zu übergeben und daher war auch bei `Def_GetWeite(4) ! "GDI32", "GetTextExtentPoint32A"` die 4 anzugeben.

Welcher Art sind nun die 4 Parameter?

Unter Profan gibt es nur 2 Möglichkeiten:

- Eine Bereichsvariable
- eine numerische Angabe.

Eine numerische Angabe heißt, dass der Wert in einer LongIntvariablen (Postfix = `&`) oder direkt als Zahl angegeben wird.

Welcher Art sind nun die 4 Parameter?

`hdc` identifies the device context bzw. handle of device context. Hier kann der handle in einer LongIntvariablen oder, wie hier, als Profan-Systemvariablen angegeben werden.

`lpstring` points to the string of text bzw. address of textstring. Immer wenn Address of oder points to steht ist ein Zeichenstring zu übergeben, daher ist hier eine Bereichsvariable anzugeben (im obigen Beispiel der Prozedur ist es `Bez#`, in der zuvor der Text übertragen wurde).

`cbstring` number of characters in string. Immer wenn number angegeben ist, ist eine Zahl zu übergeben. Entweder direkt eingesetzt oder es wird eine Longintvariable, in der sich der Wert befindet, eingesetzt. Hier muss die Anzahl der Zeichen im Text übergeben werden.

`lpsize` points to a **SIZE** structure ... auch hier ist eine Bereichsvariable anzugeben. Wie sieht aber die Struktur aus?

Wenn wir im Hilfetext auf [SIZE](#) klicken öffnet sich wieder ein Fenster:

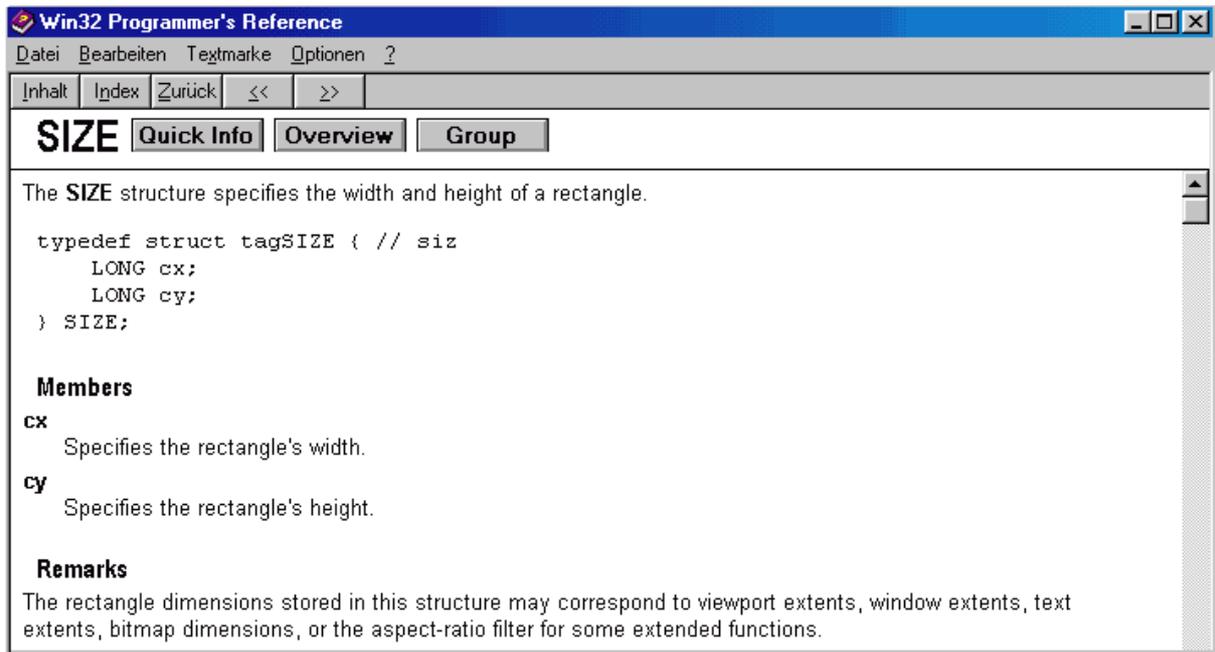


Bild 3 Aufbau der Struktur SIZE

Leider werden auch in der Hilfedatei verschiedene Begriffe verwendet. Einmal `DWORD`, `UInt`, oder wie hier `LONG`. Beide Angaben bedeuten ein 4-Byte-Wort, oder in Profan ein `LongInt (Var&)`.

Auf die Struktur bezogen heißt dies, dass in der Bereichsvariablen (in der Prozedur mit `Lng#` angegeben) 2 Longs abgelegt werden: ab `Lng#`, 0 die Breite des Rechteckes und ab `Lng#`, 4 die Höhe des Rechteckes das den zu prüfenden Text umfaßt. In der Prozedur war nur die Breite gefragt, daher wurde nur diese mit `Leng& = @Long(Lng#, 0)` in die Variable `Leng&` übernommen. Trotzdem muss die Bereichsvariable `Lng#` mit der Länge 8 DIMensioniert werden (ein `LONG` belegt 4 Bytes).

Diese Art, dass eine Bereichsvariable (im Beispiel ist es `Lng#`) nicht nur einen String oder `LongInt` beinhaltet sondern strukturiert ist, wie hier [SIZE](#), kommt sehr häufig vor.

Um wieder zur Beispielsprozedur `TextWeite` zurückzukommen:

Es wird der an die Prozedur übergebene Text als String übergeben und intern in der Prozedur mit der Variablen `T$` entgegenommen. Er wird mit der Anweisung `String` in die Bereichsvariable übertragen. Dabei wird hinter das letzte Zeichen in der Bereichsvariablen ein Nullbyte gesetzt.

Da sich [diese](#) API-Funktion aber nicht darum kümmert ob der Textstring mit einem 0-Byte abgeschlossen ist, muss der Funktion die Länge des Textes in Anzahl Zeichen mitgeteilt werden. Dazu wird `lLeng&` gesetzt.

In den meisten Fällen erwarten APIs, dass Strings mit einem Nullbyte beendet sind und können damit selbst die Stringlänge feststellen.

Nach dem API-Aufruf wird mit `Leng& = @Long(Lng#, 0)` die Textweite (in Punkten) ausgelesen. Um auch die Texthöhe festzustellen müsste noch `Hoeh& = @Long(Lng#, 4)` eingetragen werden.

Es wird noch, abhängig davon ob im 2. Parameter (`Prt_Ctl%`) der Wert 1 übergeben wurde, die Weite durch 10 dividiert (ist für die Ausgabe an den Drucker erforderlich).

Zum Vergleich zeige ich hier noch Auszüge zur selben API Funktion.

Zuerst aus „Das Win32 API“ (Band 4)



GetTextExtentPoint32

(GetTextExtentPoint32A, GetTextExtentPoint32W)
Breite und Höhe des angegebenen Strings berechnen

DLL	Index 95	Index 98	Index NT	API-Version
GDI32	205, 206	208, 209	261, 262	95/98/NT

Funktion	Parameter	Rückgabewert
C:		
<i>GetTextExtentPoint32(A,W)</i>	HDC hdc LPCTSTR (LPCSTR, LPCWSTR) lpString int cbString LPSIZE lpSize	BOOL
Delphi:		
<i>GetTextExtentPoint32(A,W)</i>	DC: HDC Str: PChar (PAnsiChar, PWideChar) Count: INTEGER VAR Size: TSize	Bool

Parameter

hdc/DC Gerätekontext.

lpString/Str Zeiger auf den String. Dieser muß nicht mit einem Nullzeichen abgeschlossen sein, da der Parameter *cbString/Count* die Länge angibt.

cbString/Count Anzahl an Zeichen im String.

lpSize/Size Zeiger auf eine Struktur des Typs *SIZE*, in der die Dimensionen des Strings zurückgegeben werden.

Die Funktion berechnet die Breite und Höhe (in logischen Einheiten) des angegebenen Text-Strings. Sie ersetzt die Funktion *GetTextExtentPoint*.

GetTextExtentPoint32 verwendet die aktuell in den Gerätekontext selektierte Schrift zur Berechnung der Dimensionen. Dabei wird angenommen, daß kein Clipping auftritt.

Da einige Geräte Zeichen unterschneiden, muß die Summe der Breiten der Zeichen nicht unbedingt der Breite des Strings entsprechen.

Die berechnete Breite enthält auch die Breite der Zwischenräume, die von der Funktion *SetTextCharacterExtra* gesetzt werden.

Ein Makro ruft automatisch statt *GetTextExtentPoint32* die richtige Funktion (*GetTextExtentPoint32A* oder *GetTextExtentPoint32W*) auf, je nachdem, ob UNICODE definiert ist.

War die Funktion erfolgreich, ist der Rückgabewert TRUE (1), im Fehlerfall hingegen FALSE (0). Erweiterte Informationen lassen sich dann mit Hilfe der Funktion *GetLastError* abrufen.

**C:**

```

void Text(HWND hFenster) {
    HDC hKontext;           // Ausgabekontext
    char acString[80] = {"Test-String"}; // Ausgabestring
    SIZE Groesse;         // Dimensionen des Strings

    hKontext = GetDC(hFenster); // Kontext bestimmen
    // ...
    // Höhe und Breite bestimmen
    if (! GetTextExtentPoint32(hKontext, acString,
                               strlen(acString), &Groesse)) {
        // Fehler
        // ...
    }
    // ...
    ReleaseDC(hFenster, hKontext); // Kontext freigeben
}

```

Delphi:

```

PROCEDURE Text(hFenster: HWND); StdCall;
VAR
    hKontext: HDC;           // Ausgabekontext
    acString: ARRAY[0..79] OF CHAR; // Ausgabestring
    Groesse : TSize;         // Dimensionen des Strings
BEGIN
    acString := 'Test-String'; // Init
    hKontext := GetDC(hFenster); // Kontext bestimmen
    // ...
    // Höhe und Breite bestimmen
    IF NOT GetTextExtentPoint32(hKontext, acString,
                                StrLen(acString), Groesse) THEN BEGIN
        // Fehler
        // ...
    END;
    // ...
    ReleaseDC(hFenster, hKontext); // Kontext freigeben
END;

```

Querverweis:

GetTextExtentPoint, *SetTextCharacterExtra*

394

Bild 5 (2. Von 2 Seiten)

Interessant ist hier, dass angezeigt wird, dass es 2 Funktionen gibt: *GetTextExtentPoint32A* und *GetTextExtentPoint32W*.

Für (ehemalige ?) Delphi- und C-Programmierer sind hier ganz gute Beispiele gegeben.

... und aus Win32 Programmierung (API Bible)

758
Drucken und Textausgabe

GETTEXTENTPOINT32
 WIN32s
 WINDOWS 95
 WINDOWS NT

Beschreibung

GetTextExtentPoint32() berechnet die Breite und Höhe eines bestimmten Strings, angegeben in logischen Einheiten. Die Funktion verwendet die aktuelle Schrift für ihre Berechnungen. Weil einige Geräte ein Kerning für bestimmte Zeichen durchführen, ist die Summe der einzelnen Zeichengrößen möglicherweise ungleich der Stringlänge. GetTextExtentPoint32() berücksichtigt Abstände zwischen den Zeichen, die bei der Größenberechnung des Strings durch SetTextCharacterExtra() gesetzt wurden.

Syntax

BOOL GetTextExtentPoint32(HDC *hdc*, LPCTSTR *lpsz*, int *cbString*, LPSIZE *lpSize*)

Parameter

<i>hdc</i>	HDC: Der Gerätekontext.
<i>lpsz</i>	LPCTSTR: Ein Zeiger auf einen Zeichenstring. Ein Endezeichen ist nicht erforderlich, weil der Parameter <i>cbString</i> die Länge des Strings angibt.
<i>cbString</i>	int: Die Anzahl der Zeichen im String.
<i>lpSize</i>	LPSIZE: Ein Zeiger auf eine SIZE-Struktur, die die Größe des Strings aufnimmt.

Rückgabewerte

BOOL: TRUE, falls die Funktion erfolgreich ausgeführt werden konnte, andernfalls FALSE.

Include-Datei

wingdi.h

Siehe auch

GetTextExtentExPoint()

Beispiel

Siehe SetTextJustification().

Bild 6

Das Ganze war ja jetzt einfach und sollte die Zusammenhänge aufzeigen, dass eine in Profan zu verwendende API-Funktion zuerst mit DEF zu definieren ist. Hier wird festgelegt welche API-Funktion benutzt wird und in welcher Windows-DLL (hier war es GDI32) sich diese befindet. Dabei ist zu beachten, dass es möglicherweise 2 Funktionen geben kann an die bei gleichem Namen entweder ein A, oder ein W anzuhängen ist. Das W trifft vermutlich vorwiegend für Asien zu weil dort wesentlich mehr Schriftzeichen existieren.

Aber nicht immer ist es so einfach.

Nehmen wir ein anderes Beispiel: die Funktion `MessageBox`. Diese Funktion ist in Profan mit der Profanfunktion `@MessageBox(S1, S2, N)` bereits vorhanden. Ich nehme dieses Beispiel um, gegenübergestellt zu Profan, die exakte Parallelität zu zeigen.

Win32 Programmer's Reference

Datei Bearbeiten Textmarke Optionen ?

Inhalt Index Zurück << >>

MessageBox Quick Info Overview Group

The **MessageBox** function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```

int MessageBox(
    HWND hWnd,           // handle of owner window
    LPCTSTR lpText,      // address of text in message box
    LPCTSTR lpCaption,   // address of title of message box
    UINT uType           // style of message box
);

```

Parameters

hWnd
Identifies the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText
Points to a null-terminated string containing the message to be displayed.

lpCaption
Points to a null-terminated string used for the dialog box title. If this parameter is NULL, the default title Error is used.

uType
Specifies a set of bit flags that determine the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

Specify one of the following flags to indicate the buttons contained in the message box:

Flag	Meaning
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_OK	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.

Specify one of the following flags to display an icon in the message box:

Flag	Meaning
MB_ICONEXCLAMATION, MB_ICONWARNING	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION, MB_ICONASTERISK	An icon consisting of a lowercase letter <i>i</i> in a circle appears in the message box.
MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	A stop-sign icon appears in the message box.

Specify one of the following flags to indicate the default button:

Flag	Meaning
MB_DEFBUTTON1	The first button is the default button. MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.
MB_DEFBUTTON2	The second button is the default button.
MB_DEFBUTTON3	The third button is the default button.
MB_DEFBUTTON4	The fourth button is the default button.

Specify one of the following flags to indicate the modality of the dialog box:

Flag	Meaning
MB_APPLMODAL	The user must respond to the message box before continuing work in the window identified by the <i>hWnd</i> parameter. However, the user can move to the windows of other applications and work in those windows. Depending on the hierarchy of windows in the application, the user may be able to move to other windows within the application. All child windows of the parent of the message box are automatically disabled, but popup windows are not. MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.
MB_SYSTEMMODAL	Same as MB_APPLMODAL except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with <i>hWnd</i> .
MB_TASKMODAL	Same as MB_APPLMODAL except that all the top-level windows belonging to the current task are disabled if the <i>hWnd</i> parameter is NULL. Use this flag when the calling application or library does not have a window handle available but still needs to prevent input to other windows in the current application without suspending other applications.

Bild 7 Beschreibung der Funktion MessageBox (Teil 1 von 2)

In addition, you can specify the following flags:

MB_DEFAULT_DESKTOP_ONLY
The desktop currently receiving input must be a default desktop; otherwise, the function fails. A default desktop is one an application runs on after the user has logged on.

MB_HELP
Adds a Help button to the message box. Choosing the Help button or pressing F1 generates a Help event.

MB_RIGHT
The text is right-justified.

MBRTLREADING
Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems.

MB_SETFOREGROUND
The message box becomes the foreground window. Internally, Windows calls the [SetForegroundWindow](#) function for the message box.

MB_TOPMOST
The message box is created with the WS_EX_TOPMOST window style.

MB_SERVICE_NOTIFICATION
Windows NT only: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer.
If this flag is set, the *hWnd* parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the *hWnd*.
For Windows NT version 4.0, the value of MB_SERVICE_NOTIFICATION has changed. See WINUSER.H for the old and new values. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of **MessageBox** and **MessageBoxEx**. This mapping is only done for executables that have a version number, as set by the linker, less than 4.0.
To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Windows NT 3.x and Windows NT 4.0, you have two choices.

1. At link-time, specify a version number less than 4.0; or
2. At link-time, specify version 4.0. At run-time, use the **GetVersionEx** function to check the system version. Then when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X; and on Windows NT 4.0, use MB_SERVICE_NOTIFICATION.

MB_SERVICE_NOTIFICATION_NT3X
Windows NT only: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51.

Return Values
The return value is zero if there is not enough memory to create the message box.
If the function succeeds, the return value is one of the following menu-item values returned by the dialog box:

Value	Meaning
IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDYES	Yes button was selected.

If a message box has a Cancel button, the function returns the IDCANCEL value if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

Remarks
When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the *lpText* and *lpCaption* parameters should not be taken from a resource file, because an attempt to load the resource may fail.
When an application calls **MessageBox** and specifies the MB_ICONHAND and MB_SYSTEMMODAL flags for the *uType* parameter, Windows displays the resulting message box regardless of available memory. When these flags are specified, Windows limits the length of the message box text to three lines. Windows does *not* automatically break the lines to fit in the message box, however, so the message string must contain carriage returns to break the lines at the appropriate places.
If you create a message box while a dialog box is present, use the handle of the dialog box as the *hWnd* parameter. The *hWnd* parameter should not identify a child window, such as a control in a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

See Also
[FlashWindow](#), [MessageBeep](#), [MessageBoxEx](#), [MessageBoxIndirect](#), [SetForegroundWindow](#)

Bild 8 Beschreibung der Funktion MessageBox (Teil 2 von 2)

Mit dem Click auf „Quick Info“ erfahren wir, es handelt sich um die User32.dll und bei Unicode findet sich ein Eintrag. Damit steht fest, dass die Funktion MessageBoxA heißt. Weiters sehen wir, dass es 4 Parameter gibt.

Somit wäre die Definition in Profan:

```
DEF Zeige_Box(4) ! "User32", "MessageBoxA"
```

Beispiel für den Aufruf.

Nehmen wir an: für den Meldungstext verwenden wir die Bereichsvariable Txt1#, für die Titelzeile Txt2#, für die Attribute (Aussehen, Buttons etc.) Atr&.

Die Funktion liefert uns die Antwort in Ret&. In diesem Falle wenn Ret& = 0 dann trat ein Fehler auf.

Ein Wert > 0 zeigt an mit welchem Button die Messagebox beantwortet wurde (siehe in Bild 8 Absatz **Return Values**)

Die Bereichsvariablen sind entsprechend der Textlängen ausreichend zu DIMensionieren:

Die Messagebox soll normale Größe, einen **Ja** und einen **Nein** Button haben, default soll der 2. Button (Nein) vorgegeben sein und es soll das Fragezeichen als Icon angezeigt werden. Nach dem Aufruf soll angezeigt werden, welcher-Button gedrückt wurde.

```
DEF Zeige_Box(4) ! "User32", "MessageBoxA"
Declare Txt1#,Txt2#,Atr&,Ret&
cls
Dim Txt1#,25
Dim Txt2#,25

String Txt1#,0 = "Meldungstext"
String Txt2#,0 = "Text in der Titelzeile"
Atr& = 292 ' = $124
Ret& = @Zeige_Box(%Hwnd,Txt1#,Txt2#,Atr&) ' statt ATR& könnte hier ebenso $124 stehen
Case (Ret& = 6) : Print "es wurde Ja gedrückt"
Case (Ret& = 7) : Print "es wurde Nein gedrückt"
WaitInput
End
```

Was hat es mit dem Wert 292, den wir in die Variable ATR& gestellt haben auf sich?

Wenn wir in der Profan-Hilfe unter @MessageBox nachsehen finden wir dort die Lösung, um die Art der MessageBox zu definieren.

Wir addieren:

```
4 .. damit ein JA und ein NEIN Button angezeigt wird
32 .. als Icon soll das Fragezeichen gezeigt werden
256 .. es soll der 2. Button als default markiert werden
das ergibt den Wert 292.
```

Nur, was haben die Profanangaben mit der API-Funktion zu tun?

Wenn wir im Bild 7 die Werte von MB_YESNO, MB_ICONQUESTION und MB_DEFBUTTON2 OR-Verknüpfen (hinter diesen Labels stehen Hex-Werte (zu finden in meinem Programm API_Hilfe) ergibt sich der Hexwert \$124, was dezimal 292 ist. Und ob bei Atr& 292 oder \$124 angegeben wird ist völlig egal.

In diesem Fall mit der Messagebox haben wir die Möglichkeit in der Profanhilfe nachzusehen.

Aber was ist mit anderen API Funktionen?

Hier können mit meiner API_Hilfe die Labels gefunden und damit der Hexwert (\$124) gefunden werden.

Aber es geht viel einfacher!

Seit Profan 7.5 gibt es auch Headerdateien. Und in denen sind ebenfalls die Werte zu den Labels zu finden. Damit kann die Zeile Atr& = auch so aussehen:

```
Atr& = @Hex$(~MB_YESNO | ~MB_ICONQUESTION | ~MB_DEFBUTTON2)
```

Das heißt es können die Labels, wie sie in der Win32.hlp angegeben sind hier angegeben werden. Es muss nur jedes Label mit dem Zeichen ~ angeführt werden und zwischen den einzelnen Labels muss das Zeichen | (<= **Altrechts+<** => eingefügt werden.

Voraussetzung ist, dass die Headerdateien am Programmfang angeführt werden.

Eine weitere Möglichkeit (unabhängig von den Headerdateien!) gibt es dass statt der Bereichsvariablen wenn es sich um einen String handelt und wenn der String als Eingabe zum API dient.
In solchen Fällen können auch Stringvariablen verwendet werden (bitte erst ab Profan 7.5!).

Dabei ist zu beachten: Wird eine Bereichsvariable beim API Aufruf angegeben so wird intern die Adresse der Bereichsvariablen übergeben. Bei einer Stringvariablen (Var\$) würden hier die ersten Stellen des String übergeben was sicher einen Absturz bringen würde. Daher muss die Adresse des Strings an den Aufruf übergeben werden.

Und damit könnte obiges Programm auch so aussehen:

' definiere die Headerdateien (es werden nicht alle gebraucht – sie belegen aber auch keinen Platz

```
$H comdlg.ph
$H commctrl.ph
$H lzexpand.ph
$H messages.ph
$H richedit.ph
$H shellapi.ph
$H Windows.ph
Decimals 0
```

```
DEF Zeige_Box(4) ! "User32", "MessageBoxA"
Declare Tx1$, Tx2$, Atr&, Ret&
cls
```

```
Tx1$ = "Meldungstext"
Tx2$ = "Text in der Titelzeile"
```

```
Atr& = @Hex$(~MB_YESNO | ~MB_ICONQUESTION | ~MB_DEFBUTTON2)
Ret& = @Zeige_Box(%Hwnd, @Addr(Tx1$), @Addr(Tx1$), Atr&)
Case (Ret& = ~IDYES) : Print "es wurde Ja gedrückt"
Case (Ret& = ~IDNO) : Print "es wurde Nein gedrückt"
WaitInput
End
```

Nochmals zur Erinnerung!

Texte dürfen niemals in einem API-Aufruf direkt angegeben werden. Ein Aufruf:

```
@Zeige_Box(%Hwnd, "Meldungstext", Txt2#, Atr&)
```

ist unzulässig und führt ziemlich sicher zum Programmabsturz. Die Verwendung von Stringvariablen statt Bereichsvariablen muss mit Vorsicht behandelt werden, Beinhaltet die Bereichsvariable eine Struktur (Bild 3 Struktur **SIZE**) kann auf keinen Fall eine Stringvariable benutzt werden!

Bleiben zuletzt noch die Messages.

Ähnlich, wie mit den API-Funktionen kann man mit @SendMessage auch vieles steuern.

Im Prinzip ist in der Profanhilfe (fast) alles darüber beschrieben. Jedoch findet man darüber hinaus in der Win32.hlp noch weitere Informationen – man muss sie nur finden.

Wir benutzen die Profanfunktion

```
@SendMessage(N1, N2, N3, N4)
```

N1 ist immer der Handle des Dialogelementes (Button, Listbox, Fenster ...)

N2 ist immer die Meldungsnummer

N3 (in Win32.hlp als wParam bezeichnet)

N4 (in Win32.hlp als lParam bezeichnet)

N2 ist ja noch eindeutig. Wobei hier der dezimale oder der Hexwert angeführt werden kann.

N3 und N4 hängen direkt mit der Meldung (und damit mit N2) zusammen.

Um ein Beispiel zu zeigen suchen wir in der Profanhilfe unter Messages / Editmessages die Messagenummer um die Anzahl der Zeichen für die Eingabe zu begrenzen und finden sie unter em_LimitText

Links daneben sind die Messagenummern \$0415 / \$00C5. Da wir unter Windows95 (aufwärts) arbeiten darf nur die zweite Nummer \$00C5 verwendet werden! Mit Click auf em_LimitText bekommen wir mehr Informationen. Unter N3

ist die Anzahl der Stellen anzuführen (direkt als Zahl oder über eine numerische Variable). N4 ist nicht angeführt, daher setzen wir 0 für N4 ein.

mit

```
Ret& = @SendMessage(EditHandle%,$00C5,7,0)
```

würden wir festlegen, dass in das Editfeld maximal 7 Zeichen eingegeben werden können. Das Ergebnis ist in diesem Fall kaum interessant, daher kann "Ret& =" auch weggelassen werden.

Es ist auch möglich statt \$00C5 oder auch \$C5 den dezimalen Wert 197 einzugeben.

Nachdem es über die verschiedenen Messages mehr bzw. umfangreichere Informationen gibt kann man in der Win32.hlp danach suchen. Entweder unter Edit Control Messages oder unter Em_ für Edits (für Listboxen Lb_, für Buttons Bm_ usw.)

Damit endet hier die Beschreibung über die Grundlagen.

Im Anhang folgen noch einige Informationen.

Anhang.

Literatur:

Die Hilfedatei **Win32.hlp** sollte bei Microsoft zu finden sein. Es gibt aber weitere Adressen (die sich leider verändern können!). Ich fand die folgende Adresse am 21.2.2008. Da immer wieder Webseiten stillgelegt werden, kann es erforderlich sein über Google (o.ä.) mit dem Suchbegriff Win32.hlp zu suchen.

Download von <http://codingcrew.de> hat eine Größe von 19,1MB (Version ?)
(suchen unter Win32.Assembler / Win32.hlp)

Win32 Programmierung (API Bible) von Richard J. Simon, deutsche Übersetzung von Judith Muhr.
Komplette Referenz zu Windows 95 und NT. Alle APIs und Kontrollfunktionen mit Beispielen.
Mit Programmbeispielen (nicht in Profan!) auf CD. Die Beispiele dürften in C++ geschrieben sein.
Haar bei München: SAMS ISBN 3-8272-4502-8

Das Win32 API in 5 Bänden. Hier werden zu allen Funktionen Beispiele in C++ und Delphi angeführt. Doch sehe ich diese Bücher nur als Referenz; umfangreicher beschrieben. Zum Erlernen ist aber die API Bible wahrscheinlich eher geeignet. Im Win32 API sind die Strukturen z.T. genauer beschrieben und oft sind nicht nur Labels sondern auch die zugehörigen Werte angeführt. In der Win32.hlp sind anzugebende Werte nur als Label (Labels, symbolische Bezeichnung) aber nie mit dem anzugebenden Wert angeführt.

Die Bücher wurden von Wolfgang Soltendick herausgegeben und erschienen im Verlag C&L.

Band 1 beschreibt „LZ32“, „ComCtl32“ und „Kernel32“. ISBN 3-932311-05-1
Band 2 beschreibt „SHELL32“, „COMDLG32“ und „OLE32“. ISBN 3-932311-36-1
Band 3 beschreibt „User32“ und „AdvAPI32“. ISBN 3-932311-37-X
Band 4 beschreibt „DiskCopy“, „GDI32“ und „MultiMedia“. ISBN 3-932311-52-2
Band 5 beschreibt ???

Sonstiges.

Hexwerte

Da immer wieder der Begriff Hexwert (oder hexadezimaler Wert) vorkam möchte ich erklären was das ist.

Der Speicher ist im PC in Speicherstellen unterteilt. Diese werden auch als Bytes bezeichnet. Jedes Byte kann ein Zeichen (Ziffern, Buchstaben, Sonderzeichen) aufnehmen.

In Profan gibt es auch noch Integer und LongInteger. Ein Integer umfasst (früher 2 Bytes, jetzt 4Bytes) ein LongInteger (auch LongInt) umfaßt 4 Bytes. Diese Zusammenfassung ändert physisch nichts, es ist nur eine logische Zusammenfassung.

Ein Byte ist noch unterteilt in 8 Bits. Ein Bit ist die kleinste Einheit. Jedes Bit stellt, abhängig von seiner Position im Byte, einen bestimmten Wert dar.

128	64	32	16	8	4	2	1	← Wert
7	6	5	4	3	2	1	0	← Bitposition

Ein Bit kann nur (aus = 0) oder (ein = 1) sein. Wenn also die Bits 0 und 3 ein sind so ist der Wert 9 gespeichert, bei Bit 2 und 4 ein ist es der Wert 6.

Für diesen Wert 6 würde man in Profan binär ausgedrückt %00000110 schreiben. Das ist sehr unübersichtlich. Oder nicht?

Daher hat man eine andere Darstellung erfunden: die hexadezimale Schreibweise. Die Wertigkeit der Bits bleibt unverändert, doch für die Darstellung teilt man das Byte in 2 Hälften und zählt bei den Bits 4 bis 7, nur für die Schreibweise, wieder 1, 2, 4, 8. Und das sieht dann so aus:

128	64	32	16	8	4	2	1	← Wert (echter Wert)
8	4	2	1	8	4	2	1	← Wert (für Hexdarstellung)
7	6	5	4	3	2	1	0	← Bitposition

Wenn jetzt wieder die Bits 1 und 2 gesetzt sind ist das der hexadezimale Wert 06 und wir schreiben dafür \$06. Ist auch das Bit 7 gesetzt so lautet der Ausdruck \$86 – was dem dezimalen Wert 134 entspricht (2+4+128).

Das ist viel praktischer den Wert so auszudrücken. So können wir pro Halbbyte bis 15 zählen. Nur, was machen wir wenn der Wert größer als 9 ist? Hier mit 10, 11, ... weiterzuzählen sieht optisch nicht gut aus. Daher zählen wir ab 10 mit den Buchstaben A bis F weiter. Während also bei einem numerischen Wert nur die Ziffern 0 bis 9 vorkommen können sind es bei einem hexadezimalen Wert die Zeichen 0 bis 9 und A bis F

Ein Wert \$4C entspricht damit dem dezimalen Wert 76 (Bit 2, 3 und 6 = 4+8+64).

Das ist das ganze Geheimnis der Darstellung von Werten in Hexadezimal. Natürlich ist das nicht ausschließlich auf 1 Byte beschränkt. Es können damit auch 2 oder 4 Bytes so dargestellt werden. Das jeweils vorangestellte Byte beginnt dann im Bit 0 mit der Wertigkeit des Bit $7 * 2$ des vorherigen Bytes. Also wenn 2 Bytes so zusammengehängt werden hat im vorderen (werthöheren) Byte das Bit 0 den Wert 256, das Bit 1 den Wert 512 und so weiter.

Der Ausdruck \$014C drückt damit den dezimalen Wert 332 aus (256 + 64 + 8 + 4).

Die Oder-Verknüpfung von Hexwerten.

Wie schon erwähnt werden bei Flags (siehe MessageBox) oft mehrere Hexwerte kombiniert. Diese müssen Oder-verknüpft werden.

Die zu verknüpfenden Werte können direkt als String oder in einer String-Variablen übergeben werden. Auch ein Mix ist zulässig.

Hier ein Beispiel:

```
Declare V2$,V3$,V4$
```

```
V2$ = "33"           ' dezimaler Wert = $21
```

```
V3$ = $12
```

```
V4$ = "$"+@Hex$($40 | V2$ | V3$ | $70010003)
```

```
MessageBox(V4$,"",0) ' muss $70010037 anzeigen
```

```
End
```

Es ist egal ob die zu verbindenden Werte über Stringvariablen oder direkt angegeben werden. Auch dezimale Werte, als **String** sind zulässig.

```
' Das Ergebnis ist $70010037 (oder dezimal 1879113783) und steht als Ergebnis in V4$.
```

```
'      Hexwert      =      Byte 3      Byte 2      Byte 1      Byte 0      = 1 LongInt&
'      $00000040 = %0000 0000 0000 0000 0000 0000 0100 0000
' or $00000021 = %0000 0000 0000 0000 0000 0000 0010 0001
' or $00000012 = %0000 0000 0000 0000 0000 0000 0001 0010
' or $70010003 = %0111 0000 0000 0001 0000 0000 0000 0011
' =====
' = $70010037 = %0111 0000 0000 0001 0000 0000 0111 0011 = Resultat
```

ACHTUNG: ist das linkeste Bit (\$80000000) gesetzt werden die Bits zwar richtig verknüpft, die Funktion @Hex\$ interpretiert dies als Minus und der Wert kann falsch sein.

Bei der @MessageBox in der Profanhilfe wurden die Werte 4, 32 und 256 (\$4, \$20 und \$100) addiert. Warum dann die Werte unbedingt ODER-Verknüpfen? Im Beispiel der @MessageBox wäre das Ergebnis gleich. Aber in dem Moment wo bei den zu verknüpfenden Werten ein bestimmtes Bit in der selben Position mehrmals vorkommt (siehe Byte 0) würde die Addition nicht \$70010037 sondern \$700100AA ergeben was aber falsch wäre.