

# **Windows-Programmierung mit C++**



Henning Hansen

# Windows-Programmierung mit C++

**eBook**

Die nicht autorisierte Weitergabe dieses eBooks  
an Dritte ist eine Verletzung des Urheberrechts!



ADDISON-WESLEY

---

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England  
Don Mills, Ontario • Sydney • Mexico City  
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme  
Ein Titeldatensatz für diese Publikation ist bei  
Der Deutschen Bibliothek erhältlich

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.  
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

**Umwelthinweis:**

Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.  
Die Einschumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

05 04 03 02 01

**ISBN 3-8273-1747-9**

© 2001 by Addison-Wesley Verlag,  
ein Imprint der Pearson Education Deutschland GmbH  
Martin-Kollar-Straße 10–12, D-81829 München/Germany  
Alle Rechte vorbehalten

Einbandgestaltung:	Vera Zimmermann, Mainz
Lektorat:	Christina Gibbs, cgibbs@pearson.de
Herstellung:	Anja Zygalakis, azygalakis@pearson.de
Satz und Layout:	mediaService, Siegen (www.mediaproject.net)
Druck und Verarbeitung:	Nørhaven, Viborg (DK)

Printed in Denmark

# Inhaltsverzeichnis

## Teil I – Start up!

<b>1</b>	<b>Einführung</b>	<b>15</b>
1.1	Allgemeines	15
1.1.1	Vorraussetzungen	15
1.1.2	Das Ziel des Buches	15
1.1.3	Die Bedienung der Compiler	15
1.2	Die erste Anwendung	18
1.2.1	Schon geht es los	18
1.2.2	Erläuterung des Quelltexts	20
1.3	Allgemeines zu den Objekten	31
1.4	Hinweis	31
<b>2</b>	<b>Einführung in die GDI</b>	<b>33</b>
2.1	Was ist die GDI?	33
2.2	Das GDI-Programm	34
2.2.1	Der Quellcode	34
2.2.2	Erläuterung des Quelltexts	37
<b>3</b>	<b>Textausgabe mit der GDI</b>	<b>49</b>
3.1	Allgemeines	49
3.2	Text mit der GDI erstellen	50
3.2.1	Der Quelltext	50
3.2.2	Die Bedeutung des Quelltexts	54
<b>4</b>	<b>Steuerelemente</b>	<b>59</b>
4.1	Allgemeines	59
4.2	Eine Anwendung mit einem Steuerelement	60
4.2.1	Erläuterung des Quelltexts	64



<b>5</b>	<b>Das Steuerelement Edit-Feld</b>	<b>73</b>
5.1	Allgemeines	73
5.2	Der Quelltext	73
5.2.1	Was ist neu?	81
<b>6</b>	<b>System-Shutdown</b>	<b>85</b>
6.1	Allgemeines	85
6.2	Eine Anwendung	85
6.2.1	Beschreibung der Anwendung	85
6.2.2	Erläuterung des Quelltexts	93
<b>7</b>	<b>Bitmaps</b>	<b>97</b>
7.1	Allgemeines	97
7.1.1	DDB	97
7.1.2	DIB	98
7.1.3	Farbtiefe	98
7.2	Die DDB-Anwendung	99
7.2.1	Der Quelltext	99
7.2.2	Besprechung des Quelltexts	103
7.3	Die DIB-Anwendung	108
7.3.1	Der Quelltext	108
7.3.2	Besprechung des Quelltexts	112
<b>8</b>	<b>Menüs</b>	<b>117</b>
8.1	Allgemeines	117
8.2	Die Anwendung	118
8.2.1	Der Quelltext	118
8.2.2	Erläuterung	126
<b>9</b>	<b>Behandlung von Dateien</b>	<b>135</b>
9.1	Allgemeines	135
9.1.1	Dateisysteme	135
9.1.2	Funktionen	135
9.2	Die Anwendung	135
9.2.1	Der Quelltext	135
9.2.2	Erläuterung	141

<b>10</b>	<b>Anwendungen, Prozesse und Threads</b>	<b>147</b>
10.1	Allgemeines	147
10.2	Eine Multi-Threading-Anwendung	147
10.2.1	Der Quelltext	147
10.2.2	Erläuterung	151
<b>11</b>	<b>DLL-Dateien</b>	<b>155</b>
11.1	Allgemeines	155
11.2	Die Anwendung	156
11.2.1	Der Quellcode der DLL-Datei	156
11.2.2	Erläuterung	160
<b>12</b>	<b>Timer</b>	<b>163</b>
12.1	Allgemeines	163
12.2	Eine Timer-Anwendung mit Timer-Nachrichten	163
12.2.1	Der Quelltext	163
12.2.2	Erläuterung	166
12.3	Eine Timer-Anwendung mit einer Timer-Funktion	167
12.3.1	Quelltext	167
12.3.2	Erläuterung	170
<b>13</b>	<b>Der Drucker</b>	<b>171</b>
13.1	Allgemeines	171
13.2	Ein Druckerprogramm	171
13.2.1	Der Quelltext	171
13.2.2	Erläuterung	176
<b>Teil II – Take that!</b>		
<b>14</b>	<b>Win32-API: Datentypen</b>	<b>183</b>
14.1	Allgemeines	183
14.2	Tabelle	183

<b>15</b>	<b>Funktionen, Strukturen, Nachrichten und Objekte der Win32-API</b>	<b>185</b>
15.1	Allgemeines	185
<b>16</b>	<b>Win32-API: Windows-Grundlagen</b>	<b>187</b>
16.1	Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die Windows-Grundlagen	187
16.1.1	Fensterobjekt	188
16.1.2	CreateWindow	188
16.1.3	WM_CREATE	193
16.1.4	CREATESTRUCT	193
16.1.5	DestroyWindow	195
16.1.6	WM_DESTROY	195
16.1.7	ShowWindow	196
16.1.8	GetMessage	197
16.1.9	PeekMessage	199
16.1.10	RegisterClass	201
16.1.11	WNDCLASS	202
16.1.12	TranslateMessage	205
16.1.13	DispatchMessage	206
16.1.14	WM_PAINT	207
16.2	Beispiele	208
16.2.1	Ein normales Fenster erstellen	208
<b>17</b>	<b>Win32-API: GDI</b>	<b>211</b>
17.1	Funktion, Strukturen, Nachrichten und Objekte der Win32-API für die GDI	211
17.1.1	Gerätekontext-Objekt	215
17.1.2	BeginPaint	216
17.1.3	EndPaint	217
17.1.4	PAINTSTRUCT	217
17.1.5	GetDC	218
17.1.6	ReleaseDC	219
17.1.7	GetWindowDC	219
17.1.8	SetPixel	220
17.1.9	GetPixel	221
17.1.10	MoveToEx	222
17.1.11	POINT	222
17.1.12	LineTo	223



17.1.13	PolyLine	224
17.1.14	PolyBezier	225
17.1.15	Rectangle	225
17.1.16	FillRect	226
17.1.17	RECT	227
17.1.18	Ellipse	227
17.1.19	CreatePen	228
17.1.20	Pen-Objekt	229
17.1.21	SelectObject	230
17.1.22	DeleteObject	231
17.1.23	CreateSolidBrush	231
17.1.24	Brush-Objekt	232
17.1.25	TextOut	232
17.1.26	SetTextColor	233
17.1.27	SetBkColor	234
17.1.28	SetTextAlign	234
17.1.29	SetBkMode	236
17.1.30	RGB	237
17.1.31	CreateRectRgn	237
17.1.32	Region-Objekt	238
17.1.33	CombineRgn	238
17.1.34	SetWindowRgn	239
17.1.35	GetStockObject	240
17.1.36	DrawText	242
17.2	Beispiele	244
17.2.1	Einen Grafikbereich in der Nachricht WM_PAINT ermitteln und etwas hineinzeichnen	244
17.2.2	Pen- und Brush-Objekte erstellen und zuweisen	247
17.2.3	Textausgabe	250
17.2.4	Regionen benutzen	253
<b>18</b>	<b>Win32-API: Dateiverwaltung</b>	<b>257</b>
18.1	Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die Dateiverwaltung	257
18.1.1	CreateFile	258
18.1.2	CloseHandle	262
18.1.3	ReadFile	262

18.1.4	WriteFile	263
18.1.5	CopyFile	264
18.1.6	DeleteFile	265
18.1.7	MoveFile	266
18.2	Beispiele	267
18.2.1	Eine einfache Datei erstellen und sie mit Daten füllen	267
18.2.2	Eine einfache Datei öffnen und Daten aus ihr lesen	267
<b>19</b>	<b>Vordefinierte Fensterklassen</b>	<b>269</b>
19.1	Allgemeines	269
19.1.1	BUTTON	270
19.1.2	EDIT	282
19.1.3	ListBox	304
19.1.4	Static	308
<b>Teil III – Go ahead!</b>		
<b>20</b>	<b>DirectX</b>	<b>313</b>
20.1	Allgemeines	313
20.2	Ein DirectX-Programm	313
20.2.1	Allgemeines	313
20.2.2	Quelltext	314
20.2.3	Beschreibung	318
20.3	DirectX und Bitmaps	320
20.3.1	Allgemeines	320
20.3.2	Quelltext	321
<b>21</b>	<b>UNICODE</b>	<b>327</b>
21.1	Allgemeines	327
21.2	Betriebssysteme	327
21.2.1	Windows 95	327
21.2.2	Windows 98	327
21.2.3	Windows NT	327
21.2.4	Windows 2000	328
21.2.5	Windows CE	328

<b>22 COM</b>	<b>329</b>	<b>/</b>
22.1 Allgemeines	329	
22.2 DirectX	329	
<b>23 Ressourcen</b>	<b>331</b>	
23.1 Allgemeines	331	
23.2 Ein Beispiel für eine Ressource	331	
23.2.1 Allgemeines	331	
23.2.2 Erläuterung	334	
<b>Stichwortverzeichnis</b>	<b>335</b>	



TEIL 1

Nitty  
Nitty  
Gritty

START UP!



# Einführung

---

## 1.1 Allgemeines

### 1.1.1 Voraussetzungen

Sie brauchen gute Kenntnisse der Sprache C++. Außerdem benötigen Sie einen C++-Compiler mit Windowsbibliotheken. Mit MS Visual C++ 6.0 und dem Borland-Compiler C++ 5.5 können alle Beispiele dieses Buches kompiliert werden. Der Borland-Compiler ist im Internet frei erhältlich und kann von der Borland Homepage ([www.inprise.com](http://www.inprise.com)) heruntergeladen werden.

### 1.1.2 Das Ziel des Buches

Sie sollen in die Lage versetzt werden, eigene Windows-Programme zu schreiben. Dazu benötigen Sie interne Kenntnisse der Windowsarchitektur. Denn nur auf diesem Weg kommen Sie zu professionellen Programmen. Dieses Buch beruht auf Windows 98. Da sich die anderen Windowsversionen nicht sehr voneinander unterscheiden, kann das hier Gesagte aber leicht auch auf andere Versionen übertragen werden.

### 1.1.3 Die Bedienung der Compiler

#### **Borland C++ 5.5**

Installieren Sie diesen Compiler auf Ihrer Festplatte im Verzeichnis C:\BCC55. Sie können alles in gleicher Weise auf ein anderes Installationsverzeichnis anwenden. Um dies zu erreichen müssen Sie anstatt C:\BCC55 das entsprechende Laufwerk und Installationsverzeichnis angeben. Das könnte z. B. D:\meins sein. Ein Problem ist, dass der Compiler keine grafische Oberfläche besitzt. Deshalb müssen Sie den Compiler unter DOS aufrufen. Benutzen Sie dazu die *MS-DOS-Eingabeaufforderung* unter Windows.

---

Gehen Sie in das Verzeichnis `C:\BCC55` auf Laufwerk `C:`. Erstellen Sie dort ein neues Verzeichnis mit dem Namen `\in` und anschließend ein weiteres Verzeichnis mit dem Namen `\out`. Das Erstellen von Anwendungen können Sie unter *MS DOS* mit dem Befehl `md Verzeichnis Name` erreichen. Im *Windows-Explorer* können Sie einfach im Menü `DATEI | NEU ORDNER` wählen und einen Namen eingeben.

- Gehen Sie zurück in das Verzeichnis `C:\BCC55` und wechseln Sie zu `C:\BCC55\IN`. Erstellen Sie in diesem Verzeichnis eine Batch-Datei mit dem Namen `compile.bat`. Die Batch-Datei erstellen Sie mit dem *MS-DOS-Editor*. Er wird aufgerufen mit `Edit`. Am Ende dieses Punkts sehen Sie den Inhalt der Batch-Datei. Sie müssen den Inhalt genau so übernehmen. Natürlich müssen Sie, falls Sie ein anderes Installationsverzeichnis gewählt haben, die entsprechenden Verzeichnisnamen ändern. Tragen Sie nun folgende Zeile in die Batch-Datei ein.

```
C:\BCC55\BIN\BCC32.EXE -tW -IC:\BCC55\INCLUDE
-LC:\BCC55\LIB -nC:\BCC55\OUT %1 %2 %3
```

- Die Parameter haben alle eine wichtige Bedeutung. Der Parameter `-tW` gibt an, dass es sich um ein Windows-Programm handelt. Der Parameter `-I...` gibt das Verzeichnis für die Include-Dateien an (z.B. für `WINDOWS.H`). Der Parameter `-L...` zeigt das Verzeichnis für die Lib-Dateien an (z.B. für `STDLIB.LIB`). Die Parameter `%1 %2 ...` geben an, dass drei Parameter, die dieser Batch-Datei übergeben werden, an den *Compiler* `BCC32.EXE` weitergeleitet werden.
- Sie erstellen mit dem *MS-DOS-Editor* Ihren eigentlichen Quellcode. Legen Sie dafür eine Datei namens `NEU1.CPP` im Verzeichnis `C:\BCC55\IN` mit dem Editor an und fügen Sie den Quellcode wie oben angegeben ein. Speichern Sie jetzt die Datei.
- Sie befinden sich immer noch im Verzeichnis `C:\BCC55\IN`. Starten Sie jetzt die Batch-Datei wie jedes andere Programm mit dem Dateinamen `NEU1.CPP` als einzigen Parameter. Ihr Programm wird nun als Windows-Programm kompiliert und gelinkt.

```
compile neu1.cpp
```



- Wechseln Sie nun in das Verzeichnis C:\BCC55\OUT. In diesem Verzeichnis befindet sich jetzt Ihr fertiges Programm als EXE-Datei. Rufen Sie es mit dem Namen NEU1.EXE auf.

## MS Visual C++ 6.0

Es ist wesentlich einfacher, mit dieser grafischen Benutzeroberfläche ein Windows-Programm zu schreiben. Aber es macht keinen Spaß, denn sie ist meiner Meinung nach unübersichtlich. Es hängt natürlich von Ihnen ab, mit welcher Oberfläche Sie gerne programmieren. Nun aber zur Programmerstellung mit dieser Oberfläche.

- Starten Sie *MS Visual C++ 6.0*.
- Wählen Sie im Menü DATEI/NEU. Klicken Sie den Reiter PROJEKTE an und gehen Sie dort - wie in der folgenden Abbildung - zum Listenpunkt Win32-Anwendung.

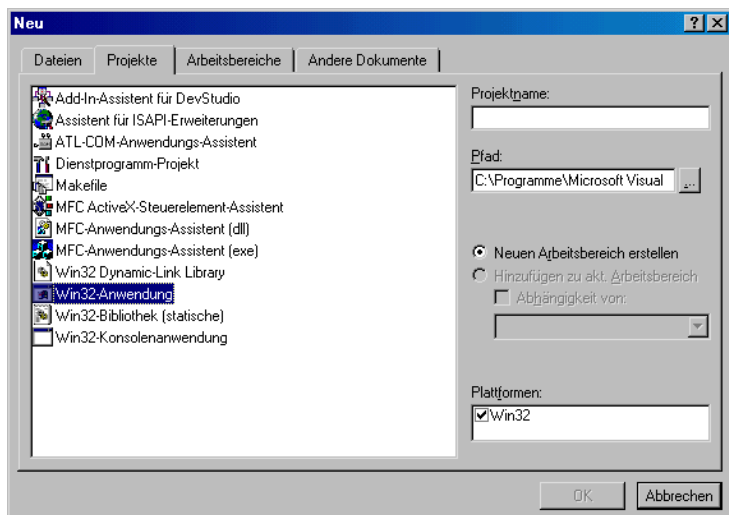


Bild 1.1: Auswahl für die Art der Anwendung

- Geben Sie einen Projektnamen *neu1* ein. Klicken Sie auf OK.
- Wählen Sie eine einfache *Win32-Anwendung* aus. Klicken Sie auf Fertigstellen.

- Gehen Sie auf OK.
- Wählen Sie am unteren linken Bildschirmrand den Reiter DATEIEN aus. Suchen Sie jetzt darüber die Datei NEU1.CPP. Klicken Sie zweimal auf diesen Dateinamen und Ihre Quellcode Datei öffnet sich. Tragen Sie hier den Quellcode ein. Sie dürfen aber die Zeilen von *WinMain* nicht noch einmal eingeben.
- Starten Sie Ihren Quellcode, indem Sie auf den MENÜPUNKT ERSTELLEN/AUSFÜHREN VON NEU1.EXE klicken.
- Speichern Sie Ihr Programm mit DATEI/ALLES SPEICHERN oder mit DATEI/SPEICHERN.
- Beenden Sie *MS Visual C++ 6.0* mit DATEI/BEENDEN.
- Ihr Programm befindet sich in *MS Visual C++ 6.0* unter dem Verzeichnis /Debug Ihres Projektes.

## 1.2 Die erste Anwendung

### 1.2.1 Schon geht es los

In unserer ersten Anwendung geht es darum, ein Fenster zu erzeugen. Dieses Fenster soll auf dem Bildschirm angezeigt und vom Benutzer geschlossen werden können. Hier sehen Sie den Quelltext für diese Anwendung.

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

```
int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
```

```
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
```

```
WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
WndClass.hCursor = 0;
WndClass.hIcon = 0;
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hWnd;
hWnd = CreateWindow("WinProg", "Fenster",
                   WS_OVERLAPPEDWINDOW,
                   0, 0, 400, 400, NULL, NULL,
                   hInstance, NULL);

ShowWindow (hWnd, nCmdShow);

UpdateWindow (hWnd);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}
return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default:
            return DefWindowProc (hWnd, uiMessage,
                                   wParam, lParam);
    }
}
```

So sollte die Anwendung aussehen, nachdem sie gestartet wurde.



Bild 1.2: Die Anwendung nach dem Start

### 1.2.2 Erläuterung des Quelltexts

Sie wissen jetzt, dass diese Anwendung ein Fenster anzeigt. Dieses Fenster kann vom Benutzer geschlossen werden, wodurch die Anwendung beendet wird. Was aber genau passiert, besprechen wir jetzt.

#### Einbindung der Datei `WINDOWS.H`

Diese Datei enthält Verweise auf andere Header-Dateien, mit denen die Windows-Funktionen, also die API-Funktionen, angesprochen werden. Außerdem befinden sich in diesen Header-Dateien (bzw. in Verweisen auf andere Header-Dateien) die Datentyp- und Strukturdefinitionen. Die API-Funktionen werden von Windows bereitgestellt.

#### Der Prototyp einer neuen Funktion

Diese Funktion ist enorm wichtig. An dieser Stelle sei nur so viel gesagt, dass sie später für die Kommunikation des Fensters mit Windows genutzt wird.

## Die WinMain Funktion

Es handelt sich um die Einsprungsfunktion, denn hier beginnt der eigentliche Code der Anwendung. Diese Funktion wird von allen Windows-Programmen verwendet.

## Die Fensterklasse

Als Erstes wird eine Struktur vom Typ `WNDCLASS` deklariert. Die Struktur bestimmt die allgemeinen Eigenschaften der Fenster, die mit ihr erzeugt werden. Sie erhält den Namen `WndClass`. Die Variablen der Struktur werden nun genauer erläutert.

```
typedef struct _WNDCLASS
{
    UINT    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;
    HICON   hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

In der Anwendung wird die `WNDCLASS` Struktur wie folgt deklariert.

```
WNDCLASS WndClass;
WndClass.cbClsExtra = 0;
WndClass.cbWndExtra = 0;
WndClass.lpfnWndProc = WndProc;
WndClass.hInstance = hInstance;
WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
WndClass.hCursor = 0;
WndClass.hIcon = 0;
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";
```

Sie wundern sich wahrscheinlich über die verschiedenen Variablentypen. Diese Typen sind in den Windows-Header-Dateien definiert. Es sind einfach andere Bezeichnungen für normale C++-Variablentypen. So ist `UINT` ein `unsigned int`, `WNDPROC` ein Zeiger auf eine Funktion, `HANDLE`, `HCURSOR`, `HBRUSH` `unsigned int` und `LPCTSTR` ein konstanter Zeiger vom Typ `char`.

Die Struktur umfasst mehrere Variablen.

- Die Variable `style` legt das allgemeine Aussehen der Fenster fest, die mit dieser Struktur erzeugt werden.
- `lpfnWndProc` ist ein Zeiger auf eine Funktion, die das Fenster benutzt, um mit Windows zu kommunizieren.
- `cbClsExtra` ist uninteressant, da es hier nur um die Reservierung für zusätzliche Bytes geht.
- `cbWndExtra` hat genau das gleiche Schicksal wie `cbClsExtra`.
- `hInstance` ist sehr wichtig. Dieser Parameter legt die Zugehörigkeit aller erzeugten Fenster zu einer Anwendung fest. Hier treffen wir zum ersten Mal auf den Variablentyp `HANDLE`. Handles sind für die Windows-Programmierung außerordentlich wichtig. Sie sind `unsigned int` und dienen als Index in eine Liste. In dieser Liste befinden sich Verweise auf Datenstrukturen, die Bestandteil des Objekts sind. Eine Funktion, die ein Objekt erstellt, liefert einen Handle zurück, der dieses Objekt eindeutig identifiziert. In der Praxis sieht das so aus: Wir rufen die Windows-Funktion `CreateWindow` auf. Diese Funktion erstellt eine Datenstruktur, nämlich das Objekt, das verschiedene Eigenschaften des Fensters enthält, z.B. aus `WNDCLASS`. Diese Struktur muss nun im Speicher gekennzeichnet sein. Das passiert durch einen Eintrag in einer Liste. Die Funktion liefert einen Handle zurück. Dieser stellt nun einen Index in diese Liste dar. Hier wird ein Handle verlangt, der unser Programm kennzeichnet.
- `hIcon` und `hCursor` sind wiederum Handles auf `Icon` und `Cursor`-Objekte.
- `hBrush` ist der Handle auf ein Objekt, das ein Füllmuster enthält. In diesem Falle wird der Handle des Objekts `WINDOW_COLOR` genommen.

- `lpzMenuName` ist ein konstanter Zeiger auf einen `char`-Array, also eine Zeichenkette. Da wir kein Menü benötigen, wird dieser Wert auf `0` gesetzt.
- `lpzClassName` ist der Name der Datenstruktur, die erzeugt wird. Diese Variable ist ein konstanter Zeiger auf einen `char`-Array. Alle Zeiger auf `char`-Arrays unter Windows müssen null-terminiert sein. Das bedeutet, dass sie mit einem Nullwert abgeschlossen sein müssen. Im ganzen Buch wird in erster Linie mit ANSI-Zeichenfolgen gearbeitet. Näheres zu UNICODE erfahren Sie in den späteren Kapiteln. Diese Datenstruktur dient später dazu, Objekte zu erzeugen. Also ist sie eine Klasse, denn die Datenstruktur beinhaltet Daten und Funktions-Definitionen. Man nennt diese Datenstruktur deswegen auch Fensterklasse.

Was bedeutet es nun aber eigentlich, einen Handle zu haben? Es existieren also Datenstrukturen im Speicher, wie z. B. die eines Fensters. Für diese Datenstrukturen existieren Funktionen, alles zusammen ist ein Objekt. Die Beschreibung, welche Funktion zu welcher Datenstruktur gehört, ist demnach eine Klasse. So gibt es Fensterklassen und Fensterobjekte. Die Fensterobjekte werden durch Handles angesprochen. Das heißt ein Handle ist ein Index in einer Liste. Durch den Index kann auf die Daten des Objekts zugegriffen werden. Dieser Zugriff erfolgt durch Funktionen für die Objekte. Den Funktionen wird der Handle übergeben. Sie arbeiten nur mit einem bestimmten Objekt und sie verändern die Daten eines Objekts.

Wie schon erwähnt, wird die Struktur `WNDCLASS` jetzt bei Windows registriert. Das heißt Windows wird über die Existenz dieser Klasse informiert, damit Sie mit weiteren Windows-Funktionen Fenster mit dieser Klasse erzeugen können. Die Funktion heißt `RegisterClass`.

```
ATOM RegisterClass (CONST WNDCLASS *lpWndClass);
```

Sie wird in diesem Programm wie folgt aufgerufen.

```
RegisterClass(&WndClass)
```

Die Funktion verlangt den Zeiger auf die Struktur und gibt einen `Return` wieder. Dieser `Return` ist `null`, wenn die Funktion fehlschlägt. Ansonsten interessiert uns der `Return` im Moment nicht.

## Erstellen des Fensters

Als Nächstes folgt nun die Erstellung des Fensters. Verwenden Sie dazu die Funktion `CreateWindow`. Zuvor müssen Sie aber eine Variable deklarieren. Diese Variable muss vom Typ `HWND` sein. Dieser Typ wiederum ist ein Handle für ein Fenster.

```
HWND hWnd;
```

Nun wird die Funktion `CreateWindow` aufgerufen. Sie erstellt eine Datenstruktur für das Fenster, also das Objekt. Diese Datenstruktur weiß z.B., welche Anwendung das Fenster erzeugt hat und welche Funktion zur Kommunikation mit Windows aufgerufen werden soll. Die Funktion `CreateWindow` legt außerdem die spezifischen Eigenschaften des Fensters fest, die noch zu den Eigenschaften der Fensterklasse hinzukommen. Jedes Fenster kann nur durch eine Fensterklasse erzeugt werden.

```
HWND CreateWindow(LPCTSTR lpClassName,
                  LPCTSTR lpWindowName,
                  DWORD dwStyle,
                  int x,
                  int y,
                  int nWidth,
                  int nHeight,
                  HWND hWndParent,
                  HMENU hMenu,
                  HANDLE hInstance,
                  LPVOID lpParam
);
```

Jetzt wird wieder der Aufruf angezeigt.

```
hWnd = CreateWindow("WinProg", "Fenster",
                   WS_OVERLAPPEDWINDOW,
                   0, 0, 400, 400, NULL, NULL,
                   hInstance, NULL);
```

Nun zu den einzelnen Funktionsparametern:

- Der Funktionsparameter `lpClassName` ist ein konstanter Zeiger auf einen `char`-Array. Diese Zeichenkette enthält den Namen der



Fensterklasse, mit der das Fensterobjekt erzeugt werden soll. Durch die Fensterklasse wird nicht die Zugehörigkeit eines Fensters zu einer Anwendung angegeben.

- Der Funktionsparameter `lpWindowName` ist ebenfalls ein konstanter Zeiger auf einen `char`-Array. Er enthält eine Zeichenkette, die dem Fenster einen Namen gibt. Dieser Name wird in der Titelleiste des Fensters abgebildet.
- `dwStyles` enthält weitere spezifische Eigenschaften des Fensters. In diesem Fall ist es die Eigenschaft `WS_OVERLAPPEDWINDOW`. Es können mehrere Eigenschaften angegeben werden. Diese werden dann über ein logisches `or` verknüpft. In der Praxis könnte das so aussehen: `WS_Eigenschaft1 | WS_Eigenschaft2`. Beide Werte sind Konstanten. In diesen Konstanten sind die Bits so gewählt, dass jeweils nur ein einziges Bit auf 1 gesetzt ist. Das logische `or` von zwei Konstanten würde also verschiedene Bits setzen, die von der Funktion ausgewertet werden. Dieses Prinzip gilt bei vielen Windows-Funktionen, die mit dieser Art von Eigenschaften arbeiten.
- `x`, `y`, `nWidth` und `nHeight` sind die Position und Größe des Fensters. Dabei geben `x` und `y` die obere linke Position des Fenster zu ihrem übergeordnetem Fenster an. Dieser Wert wird in Pixeln berechnet. Der Wert der oberen linken Ecke ist `x=0` und `y=0`. Mit `nWidth` und `nHeight` werden die Breite und Höhe des Fensters angegeben.
- `hWndParent` ist der Handle des übergeordneten Fensters. Wenn hier 0 übergeben wird, hat das Fenster kein übergeordnetes Fenster.
- `hMenu` ist ein Handle auf ein Fenstermenü. Da im Moment noch kein Menü benötigt wird, wird der Wert auf 0 gesetzt.
- `hInstance` ist der Handle auf unsere Anwendung. Dieser Handle wird aus der `WinMain`-Funktion übergeben. Auch unsere Anwendung bekommt beim Start einen Handle von Windows zugewiesen. Da auch für die Datenstruktur Ihrer Anwendung, die nicht nur aus dem Code, sondern auch aus weiteren Informationen besteht, Funktionen bestehen, kann auch Ihre Anwendung als Objekt betrachtet werden. Somit ist das Fenster der Anwendung eindeutig zugeordnet.

- Der Parameter `lParam` dient zur Nachrichtenbearbeitung unter Windows, auf die wir weiter unten eingehen.

## Anzeigen des Fensters

Jetzt muss noch das Fenster angezeigt werden. Dazu bedienen Sie sich der Funktion `ShowWindow`.

```
BOOL ShowWindow (HWND hWnd,int nCmdShow);
```

Und nun zur Verwendung im Beispielprogramm:

```
ShowWindow (hWindow, nCmdShow);
```

- Der erste Funktionsparameter `hWnd` ist der Handle des Fensters, das sichtbar gemacht werden soll.
- `nCmdShow` ist ein `int`-Wert, der verschiedene Eigenschaften der Darstellungsweise angibt. Für das Fenster wird die Darstellungsweise genommen, die uns die Funktion `WinMain` liefert.

Mit diesem Befehl wird nur das von Windows verwaltete Fenster mit seinen Eigenschaften angezeigt. Wir möchten aber, dass auch der Inhalt angezeigt wird. Als Inhalt bezeichnet man die weiße Fläche, die Sie im Bild sehen können. Für die Anzeige des Inhalts benutzt man die Funktion `UpdateWindow`.

```
BOOL UpdateWindow ( HWND hWnd);
```

Die Funktion sieht im Beispielprogramm wie folgt aus.

```
UpdateWindow (hWindow);
```

- Der erste und einzige Funktionsparameter ist der Handle des Fensters, von dem der Inhalt angezeigt werden soll.

## Die Message-Loop

Die *Message-Loop* ist einer der Hauptbestandteile von Windows-Anwendungen. Man muss wissen, wie Windows überhaupt funktioniert. Anwendungen unter Windows werden nicht mehr so ausgeführt wie Anwendungen unter DOS. Das bedeutet, dass ihr eigentlicher Code nicht mehr in einer bestimmten Reihenfolge durchlaufen wird. Da Windows ja ein Multitasking- Betriebssystem ist, sollen mehrere Anwendungen gleichzeitig ausgeführt werden können. Das geht aber nicht, denn normalerweise hat ein Rechner auch

nur einen Prozessor. Dieser kann natürlich nur jeweils einen Code ausführen. Um aber den Eindruck zu erwecken, dass alle Anwendungen nebeneinander laufen, benutzt Windows eine spezielle Technik. Dabei wird zwischen den verschiedenen Codeteilen hin- und hergewechselt und jeder Code nur eine gewisse Zeit lang ausgeführt. Eine weitere Besonderheit: Jeder Anwendung unter Windows wird eine Warteschlange zugeteilt. Das ist einfach eine Liste, in der so genannte Nachrichten eingetragen werden. Diese Nachrichten können z.B. Mausclicks, Fensterbewegungen etc. sein. Das Ganze läuft folgendermaßen: Windows stellt einen Mausclick in einem Fenster fest. Daraufhin wird eine Nachricht an die Warteschlange der entsprechenden Anwendung gesendet. Diese Nachricht enthält den Handle des Fensters, in welchem der Mausclick stattgefunden hat, die Koordinaten des Klicks, die Tasten und weitere Angaben. An dieser Stelle tritt die *Message-Loop* in Aktion. Als *Message-Loop* bezeichnet man die `while`-Schleife, die in der Beispielanwendung folgt. Die *Message-Loop* hat also die Aufgabe, Nachrichten abzarbeiten. Nachrichten werden vom System, der Anwendung oder anderen Anwendungen an Fenster gesendet. Das Prinzip der Nachrichten soll es ermöglichen, mehrere Anwendungen, die nebeneinander ausgeführt werden, auf dem Bildschirm grafisch darzustellen. Die Fenster senden sich so gegenseitig Nachrichten, um ein anderes Fenster darüber zu informieren, dass es sich neu zeichnen muss. Wenn dieses Prinzip nicht eingeführt worden wäre, könnte entweder nur jeweils eine Anwendung zu einer Zeit auf dem Bildschirm dargestellt werden, oder die Anwendungen würden ständig an dieselbe Stelle des Grafikspeichers zeichnen.

```
MSG Message;  
while (GetMessage(&Message, NULL, 0, 0))  
{  
    DispatchMessage(&Message);  
}
```

Diese Schleife kann auch in anderer Form mit folgenden Funktionen auftreten. Die Funktion `GetMessage` holt eine Nachricht aus der Warteschlange der Anwendung. Wenn hier die Nachricht `WM_QUIT` auftaucht, liefert `GetMessage` `0` zurück und die Anwendung wird beendet, weil das Programm am Ende seines Codes ist. Die Nachricht

wird in einer Nachrichten-Struktur abgespeichert. Diese Struktur hat hier den Namen `Message`. Diese `Message`-Struktur wird durch `DispatchMessage` an die Funktion des Fensters weitergegeben. Bei dem Fenster aus der Beispielanwendung bedeutet dies Folgendes: Die Funktion für die Nachrichtenverarbeitung wurde in der Fensterklasse festgelegt. `DispatchMessage` ermittelt die Funktion aus der Datenstruktur des Fensterobjekts und übergibt ihr die Nachricht zur Abarbeitung. Die Fortsetzung des Programmcodes beginnt also mit der Funktion, die in der Fensterklasse angegeben wurde. Es folgt die genaue Funktionssyntax.

```
BOOL GetMessage( LPMSG lpMsg, HWND hWnd,
                UINT wParamFilterMin, UINT wParamFilterMax);
LONG DispatchMessage( CONST MSG *lpmsg);
```

Die Funktionen wurden in der Beispielanwendung wie folgt benutzt.

```
GetMessage(&Message, NULL, 0, 0)
DispatchMessage(&Message);
```

Als Erstes wird `GetMessage` besprochen.

- Der erste Funktionsparameter `lpMsg` verlangt einen Zeiger auf eine `Message`-Struktur. Der Variablentyp `LPMSG` ist nichts anderes als ein Zeiger auf eine `MSG`-Struktur.
- Der zweite Funktionsparameter gibt den Handle des Fensters an, von welchem die Nachrichten abgearbeitet werden sollen. Mit dem Wert 0 werden alle Nachrichten abgearbeitet. Daran erkennt man, dass eigentlich nur Fenster Nachrichten empfangen können.
- Mit den Funktionsparametern `wMsgFilterMin` und `wMsgFilterMax` kann man die Reichweite der Nachrichten, die zurückgeliefert werden sollen einstellen. Diese Werte werden auf 0 gesetzt, damit alle Nachrichten zurückgeliefert werden.
- Der `return` der Funktion ist 0 bei der Nachricht `WM_QUIT` und ungleich 0 bei den anderen Nachrichten. Alle Nachrichten werden mit `WM` bezeichnet. Diese Nachricht stellen konstante Werte dar.

Als Nächstes schauen wir uns den Nachrichtenparameter von `DispatchMessage` an.

- Diese Funktion verlangt als einzigen Parameter den Zeiger auf die Nachrichtenstruktur, die der aufzurufenden Funktion übergeben werden soll.

Die Message-Struktur sieht folgendermaßen aus:

```
typedef struct tagMSG
{
    HWND    hwnd;
    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT   pt;
} MSG;
```

Diese Struktur enthält mehrere Variablen.

MSG Message

Dies sind die Variablen im Einzelnen:

- `hwnd` enthält den Handle des Fensters für den die Nachricht bestimmt ist.
- `message` enthält den Wert der Nachricht. Dieser Wert ist eine der Konstanten in der Form `WM_CREATE`, `WM_SETTEXT`,... .
- `wParam` ist der erste Funktionsparameter.
- `lParam` ist der zweite Funktionsparameter.
- `time` enthält die Entstehungszeit der Nachricht.
- `pt` ist eine weitere Struktur.

```
typedef struct tagPOINT
{
    LONG x;
    LONG y;
} POINT;
```

Die beiden Variablen der Struktur `x,y` enthalten die Position des Cursors auf dem Bildschirm.

## Der Rückgabewert der Anwendung

Die Anwendung liefert einen Rückgabewert. Dieser Rückgabewert wird aus dem Parameter der Nachricht `WM_QUIT` übernommen.

## Die Funktion zur Nachrichtenabarbeitung

Ein Fenster benötigt eine Funktion, um die an das Fenster gesendeten Nachrichten abzuarbeiten. Der folgende Prototyp der Funktion soll dabei als Ausgangsbasis dienen.

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

Die Funktionsparameter werden mit der Nachricht gefüllt.

- Der erste Parameter ist der Handle des Fensters, an das die Nachricht geht.
- Der zweite Parameter bestimmt die Nachricht.
- Der dritte und vierte Parameter sind einfach weitere Informationen oder Parameter der Nachricht.
- Der `return` dieser Funktion wird von `DispatchMessage` zurückgegeben.

In dieser Funktion werden die Nachrichten abgearbeitet. Dies geschieht durch eine Switch-Verzweigung.

```
switch(uiMessage)
{
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc (hWnd, uiMessage,
                               wParam, lParam);
}
```

Die einzige Nachricht, die von dem Beispielprogramm ausgewertet wird, ist die `WM_DESTROY`. Wenn diese Funktion gesendet wird, wird die Funktion `PostQuitMessage` aufgerufen. Diese Funktion sendet die Nachricht `WM_QUIT` an die eigene Warteschlange und beendet damit das Programm. Die Nachricht `WM_DESTROY` wird ausgelöst, wenn das Fenster geschlossen wird. Ansonsten wird die Funktion

`DefWindowProc` aufgerufen, die wiederum eine Standardfunktion für die Nachricht auslöst.

```
VOID PostQuitMessage( int nExitCode);  
LRESULT DefWindowProc( HWND hWnd, UINT Msg,  
                      WPARAM wParam, LPARAM lParam);
```

Die Verwendung der Funktionen wurde schon gezeigt. Nun gehen wir auf den Parameter von `PostQuitMessage` ein.

- `nExitCode` gibt den Wert für `wParam` in `WM_QUIT` an.

Die Parameter von `DefWindowProc` sehen folgendermaßen aus:

- Dieser Funktion werden einfach alle Parameter der Nachricht übergeben, die auch der Funktion `WndProc` übergeben wurden.

### 1.3 Allgemeines zu den Objekten

Objekte sind Hauptbestandteil der Windows-Programmierung. Sie sind keine C++-Objekte sondern Datenstrukturen im Speicher. Für die verschiedenen Datenstrukturen gibt es bestimmte Funktionen, mit denen die Bedingungen für ein Objekt erfüllt sind. Ein Fenster-Objekt kann zum Beispiel nur mit Funktionen für ein Fenster-Objekt bearbeitet werden. Eine solche Funktion wäre zum Beispiel `SetWindowPos`. Die Funktionen erwarten einen Handle. Dieser Handle ist ein Index in eine Liste. Diesen Index benötigen die Funktionen, um die Datenstruktur bearbeiten zu können. Alle erstellten Objekte sollten gelöscht werden, wenn ein Programm beendet wird.

### 1.4 Hinweis

Objekte sollten am Ende des Programms immer wieder aus dem Speicher entfernt werden. Dies erreicht man durch entsprechende Funktionen, die im SDK zu finden sind.

In diesem Buch machen wir das nicht immer, damit wir uns in der Programmierung auf das Wesentlichste konzentrieren können. Diese Codeteile kann der Programmierer später noch einfügen.





# 2 Einführung in die GDI

## 2.1 Was ist die GDI?

Die GDI ist eine Untergruppierung der Win32-API. Die Win32-API bezeichnet alle Funktionen für das 32-Bit Betriebssystem Windows. API steht für *Application Programming Interface*.

Welche Funktionen enthält die GDI nun aber genau? Die Abkürzung GDI steht für *Graphics Device Interface*. Sie enthält die Funktionen, mit denen man in ein Fenster zeichnen und schreiben kann, also die Funktionen für die Grafikausgabe in den Fenstern. Die GDI-Funktionen werden aber nicht nur auf Grafikbereiche von Fenstern angewendet. Sie sind vielmehr eine Sammlung von Funktionen, mit denen man Grafik durch ein Gerätekontext<sup>1</sup>-Objekt ausgeben kann. Durch dieses Gerätekontext-Objekt hat man keinen vollen Zugriff auf den Grafikspeicher –genauer gesagt werden die GDI-Funktionen fast nur auf den Bereich angewendet, der in der letzten Anwendung im Fenster weiß war. Man nennt diesen Bereich den Clientbereich eines Fensters. Dieser Bereich hat ein eigenes Gerätekontext-Objekt, das im Fensterobjekt angegeben ist. Mit den Funktionen `BeginPaint` und `GetDC` bekommt man einen Handle auf den Gerätekontext. Dieser bezieht sich im Falle eines Fensters auf eine Bildschirmausgabe in dem Fenster, er kann sich aber genauso gut auf eine Druckerausgabe beziehen. Also kurz gesagt: die GDI umfasst alle Funktionen, mit denen man auf den Gerätekontext zugreifen kann. Dieser muss dabei nicht ein Zugriff auf den Grafikspeicher, sondern kann genau so gut ein Zu-

---

1. Zuerst sollte man den Begriff Gerätekontext klären. Fast alle Funktionen der GDI verwenden ein Gerätekontext-Objekt. Dieses Objekt verweist auf einen Teil des Bildschirmspeichers. Diesen Teil des Speichers nennt man Gerätekontext. In diesen zeichnen die GDI-Funktionen dann. Jedes Fenster besitzt ein Gerätekontext-Objekt. Dabei beschränkt sich der Gerätekontext aber nur auf den Inhalt des Fensters. Rahmen und Titelleiste z.B. gehören nicht dazu.

griff auf die Druckerausgabe sein. Das ermöglicht es, alle Funktionen der GDI für die Grafikausgabe und die Druckerausgabe zu verwenden. Die GDI-Funktionen werden aber auch im Zusammenhang mit anderen Geräten verwendet, die etwas mit Grafik zu tun haben. Die GDI hat den Nachteil, dass der Grafikspeicher nicht direkt, sondern nur über GDI-Funktionen verändert werden kann.

Wenn man den Handle für ein Gerätekontext-Objekt ermittelt hat, setzt man ihn mit den Funktionen `EndPaint` und `ReleaseDC` wieder frei.

Die nächste Anwendung zeigt die Verwendung der GDI. In dieser Anwendung sollen verschiedene Grafikfunktionen zum Einsatz kommen. Die Ausgabe von Text ist etwas schwieriger und wird hier noch nicht besprochen. Der Einsatz der Grafikfunktionen wird begleitet von vielen Objekten, die zunächst angelegt werden müssen. So muss z.B. für eine Linie zuerst ein Pen, also ein Stift-Objekt, bereitgestellt werden. Dieses Objekt enthält dann die Informationen über die Breite, die Art und die Farbe der Linie. An dieser Stelle sei nochmals darauf hingewiesen, dass diese Objekte keine Objekte im Sinne von C++, sondern windows-eigene Objekte sind. Sie werden deshalb als Objekte bezeichnet, weil eine Datenstruktur im Speicher ist, die über `Handles` angesprochen wird. Weiterhin stehen für diese Datenstrukturen Funktionen zur Verfügung: somit handelt es sich also um ein Objekt. Auf diese Objekte können Sie nur über die Win32-API zugreifen.

## 2.2 Das GDI-Programm

### 2.2.1 Der Quellcode

Diese Anwendung soll eine Einführung in die GDI-Funktionen bieten. Hier werden speziell die GDI-Funktionen zum Zeichnen dargestellt. Die Anwendung nach dem Start sehen Sie in Abbildung 2.1.

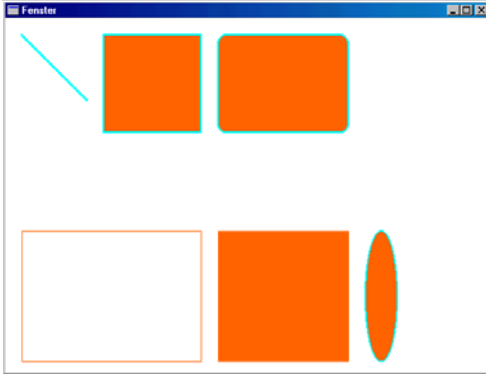


Bild 2.1: Beispielanwendung mit GDI-Zeichenfunktionen nach dem Start

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfnWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
```

```

hWindow = CreateWindow("WinProg","Fenster",
                      WS_OVERLAPPEDWINDOW,
                      0,0,600,460,NULL,NULL,
                      hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}
return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_PAINT:
            HPEN hPen;
            HPEN hPenAlt;
            HBRUSH hBrush;
            HBRUSH hBrushAlt;
            hBrush = CreateSolidBrush (RGB(255,100,0));
            hPen = CreatePen (PS_SOLID,2,RGB(0,255,255));
            HDC hdc;
            PAINTSTRUCT ps;
            hdc = BeginPaint (hWnd, &ps);
            hBrushAlt = SelectObject (hdc, hBrush);
            hPenAlt = SelectObject (hdc, hPen);
            MoveToEx (hdc, 20, 20, NULL);
            LineTo (hdc, 100, 100);
            Rectangle (hdc, 120, 20, 240, 140);
            RoundRect (hdc, 260, 20, 420, 140, 20, 20);
    }
}

```

```

RECT rect;
SetRect (&rect, 20, 260, 240, 420);
FrameRect (hdc, &rect, hBrush);
SetRect (&rect, 260, 260, 420, 420);
FillRect (hdc, &rect, hBrush);
Ellipse (hdc, 440, 260, 480, 420);
SelectObject (hdc, hBrushalt);
SelectObject (hdc, hPenalt);
DeleteObject (hPen);
DeleteObject (hBrush);
EndPaint (hWnd, &ps);
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}

```

## 2.2.2 Erläuterung des Quelltexts

### Cursor und Icon

Der Quelltext wurde in seiner Struktur so belassen wie in der ersten Beispielanwendung. Die erste Änderung zeigt sich in der Deklaration der Variablen der WNDCLASS-Struktur. Die Variable `Hcursor` bekommt einen Handle zugewiesen. Die Funktion `LoadCursor` verbindet das Cursorobjekt mit einem vordefinierten Windows-Cursor. Die Funktion `LoadIcon` bewirkt dasselbe. Sie verbindet einen vordefiniertes Icon mit einem Iconobjekt. Der Handle des Icon-Objekts wird dann genauso wie der Handle des Cursor-Objekts als `return` zurückgegeben. Schauen wir uns zunächst die Funktion `LoadCursor` an.

```

HCURSOR LoadCursor( HINSTANCE hInstance,
                    LPCTSTR lpCursorName);

```

Die Funktion wird im Quelltext folgendermaßen verwendet.

```

WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);

```

Dies sind die Funktionsparameter:

- `hInstance` wird auf `NULL` gesetzt, um einen vordefinierten Cursor zu laden.
- `lpCursorName` enthält den Bezeichner für den vordefinierten Cursor.
- Als `return` wird der Handle des Objekts wiedergegeben, mit dem der vordefinierte Cursor verknüpft ist.

Nun zur Funktion `LoadIcon`:

```
HICON LoadIcon( HINSTANCE hInstance, LPCTSTR lpIconName);
```

Die Funktion wird im Quelltext so verwendet:

```
WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
```

Die Funktionsparameter haben folgende Bedeutungen:

- `hInstance` wird auf `NULL` gesetzt, um ein vordefiniertes Icon zu verwenden.
- `lpIconName` enthält den Bezeichner für das vordefinierte Icon.
- Als `return` liefert die Funktion den Handle des Icon-Objekts, der mit dem Icon verknüpft ist.

Die beiden Funktionen in der `WNDCLASS`-Struktur sorgen dafür, dass der Cursor im Bereich des Fensters die Pfeilform annimmt. Ab sofort besitzt das Fenster auch ein Icon.

### Weitere Änderungen bis zur Message-Loop

Weitere Änderungen sind die veränderte Breite und Höhe des Fensters. Insgesamt ist das Fenster etwas größer geworden.

### Die Änderungen in der WndProc

Die wichtigste Änderung in der `WndProc` ist die Behandlung der `WM_PAINT`-Message. Teile des Gerätekontexts des Fensters können ungültig werden, weil z.B. ein anderes Fenster sie überlappt. Die Titelleisten und der Rand des Fensters werden von `DefWindowProc` selber neu gezeichnet. Wenn aber ein Teil des Gerätekontexts auf dem Bildschirm verändert wird, löst dies eine `WM_PAINT` Nachricht aus. Diese Nachricht wird der entsprechenden Fensterfunktion über `Dis-`

`patchMessage` zugesandt, bzw. die Funktion `WndProc` wird mit der Nachricht `WM_PAINT` aufgerufen. Durch die Funktion `BeginPaint` wird der ungültige Bereich eines Gerätekontexts mit `hBrush` in `WndClass.hbrBackground` gefüllt. Dann wird mit einigen Grafikfunktionen wie `LineTo` auf dem Gerätekontext gezeichnet. Zum Schluss muss der von `BeginPaint` ermittelte Gerätekontext des Fensters wieder freigegeben werden.

*Die einzelnen Eigenschaftsobjekte für die Zeichenfunktionen werden erstellt*

Als Erstes müssen für den Gerätekontext des Fensters Objekte erstellt werden, die von den einzelnen Zeichenfunktionen für das Fenster genutzt werden. Dazu werden vor allem die Funktionen `CreateSolidBrush` und `CreatePen` verwendet. `CreateSolidBrush` erstellt ein Füllmuster-Objekt und `CreatePen` ein Zeichenstift-Objekt. Dazu müssen erst einmal zwei Handles deklariert werden. Diese Handles sind vom Variablentyp `HBRUSH` und `HPEN`. Sie zeigen über einen Index in einer Liste indirekt auf die Datenstrukturen der Liste.

Für die Erstellung eines Brush-Objekts benötigt man die Funktion `CreateSolidBrush`.

```
HBRUSH CreateSolidBrush( COLORREF crColor);
```

Die Funktion wird im Quelltext folgendermaßen verwendet:

```
hBrush = CreateSolidBrush( RGB(255,100,0));
```

Der Funktionsparameter:

- `crColor` ist ein Funktionsparameter vom Variablentyp `COLORREF`. Dieser Variablentyp ist ein 32-Bit-Wert und gibt den Wert einer RGB-Farbe an. Eine RGB-Farbe setzt sich aus drei Farbanteilen zusammen. Die Farben sind Rot, Grün und Blau. Sie haben Werte zwischen 0-255 und sind, da sie jeweils nur ein Byte benötigen, alle in einem 32-Bit-Wert abgespeichert. Dieser Wert ist vom Typ `COLORREF` und wird vom Makro `RGB` erstellt. Man kann ihn auch selbst angeben, indem man die einzelnen Bytes in Hexadezimal angibt. `0x0000ffff` ist ein Beispiel für die Farbe Blau. Es werden jeweils die drei kleinsten Bytes des 32-Bit-Werts belegt.

- Als `return` gibt die Funktion einen Handle auf ein neu angelegtes Brush-Objekt zurück.
- Das RGB-Makro erstellt den `COLORREF` Wert aus Rot-, Grün- und Blauanteilen.

```
COLORREF RGB( BYTE bRed, BYTE bGreen, BYTE bBlue);
```

Der Aufruf erfolgt- wie oben beschrieben - in der Funktion `CreateSolidBrush`. Als Nächstes wenden wir uns den Parametern dieses Makros zu.

- Alle drei Parameter sind vom Typ `BYTE`, also 8-Bit große unsigned-Variablen. Sie können Werte von 0-255 annehmen. Jeder Parameter gibt den Wert des Farbanteils für die Farbe an. Der erste steht für Rot, der zweite für Grün und der dritte für Blau.

Nun zur Funktion `CreatePen`. Sie erstellt eine Pen-Objekt. Es enthält Daten für diverse Zeichenfunktionen. Meistens wird dieser Wert für Linien benutzt.

```
HPEN CreatePen( int fnPenStyle, int nWidth,
                COLORREF crColor);
```

Die Funktion wird im Quelltext so verwendet:

```
hPen = CreatePen (PS_SOLID,2,RGB(0,255,255));
```

Es folgt die Besprechung der Funktionsparameter.

- `fnPenStyle` gibt die Art der Linie an. Die Linie kann durchgehend, gestrichelt usw. sein. Der Wert `PS_SOLID` gibt eine durchgezogene Linie an. Hier werden einfach vordefinierte Werte eingesetzt, auf die die Funktion entsprechend reagiert.
- `nWidth` gibt die Breite der Linie an, die mit diesem Objekt gezeichnet wird.
- `crColor` ist wieder ein Wert vom Variablentyp `COLORREF`. Das bedeutet, es wird eine RGB-Farbe erwartet. Diese Farbe bekommt die Linie, die mit diesem Pen-Objekt gezeichnet wird. Natürlich wird auch dieser Wert mit dem RGB-Makro erzeugt.



## Die Funktionen `BeginPaint` und `EndPaint`

Beide Funktionen sind sehr wichtig. `BeginPaint` hat mehrere Auswirkungen. Wenn sie aufgerufen wird, wird der ungültige Teil des Fensters mit dem Brush-Objekt `hbrBackground` des Fensters gefüllt. Dieses Objekt wurde in `WNDCLASS` angegeben. Es war ein vordefiniertes Windows-Brush-Objekt. Danach wird der ungültige Bereich für gültig erklärt. Außerdem werden Werte in eine Struktur vom Typ `PAINTSTRUCT` zurückgegeben. Die Struktur enthält Informationen über das ungültige Rechteck. Als `return` gibt die Funktion den Handle auf ein Gerätekontext-Objekt zurück. Dieser Wert wird von allen GDI-Funktionen benötigt, denn nur in dieses Objekt kann gezeichnet werden. `BeginPaint` muss in der Funktion `WM_PAINT` stehen, denn sie erklärt das ungültige Rechteck wieder für gültig. Ein ungültiges Rechteck bezieht sich immer auf ein Gerätekontext-Objekt. Wenn eine `WM_PAINT`-Nachricht abgearbeitet und aus der Warteschlange der Anwendung entfernt wird, aber das ungültige Rechteck nicht für gültig erklärt, wird immer wieder eine `WM_PAINT`-Nachricht in der Warteschlange vorhanden sein. Falls schon eine solche Nachricht vorhanden ist, und eine weitere `WM_PAINT`-Nachricht an das Fenster gesendet werden soll, werden beide vorher zu einer Nachricht zusammengefasst. Die Beschreibung der ungültigen Rechtecke steht im Fenster-Objekt. Diese Beschreibung wird erweitert. Zum Schluss wäre noch zu erwähnen, dass jeder Handle auf ein Gerätekontext-Objekt durch die Funktion `EndPaint` auch wieder freigegeben werden muss, damit man erneut auf das Gerätekontext-Objekt zugreifen kann. Nun aber zur Funktionssyntax:

```
HDC BeginPaint( HWND hwnd, LPPAINTSTRUCT lpPaint);
```

Im Quelltext sieht das folgendermaßen aus:

```
hdc = BeginPaint (hwnd, &ps);
```

Und das sind die einzelnen Funktionsparameter:

- `hwnd` gibt den Handle auf das Fensterobjekt an, welches das Gerätekontext-Objekt enthält, in das gezeichnet werden soll.
- `lpPaint` muss ein Zeiger auf eine Struktur vom Typ `PAINTSTRUCT` sein. Diese Struktur wird mit Angaben über das ungültige Rechteck gefüllt.

- Als `return` wird ein Handle auf das Gerätekontext-Objekt zurückgeliefert. Dieser Handle wird verwendet, um mit GDI-Funktionen in das Gerätekontext-Objekt zu zeichnen.

Die Struktur `PAINTSTRUCT` hat folgenden Aufbau:

```
typedef struct tagPAINTSTRUCT
{
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

und folgende Variablen:

- `hdc` bekommt den Handle auf das Gerätekontext-Objekt zugewiesen. Es ist derselbe Handle, der von `BeginPaint` zurückgegeben wird.
- `fErase` ist `NULL`, wenn der Bereich des ungültigen Rechtecks mit `hbrBackground` aus `WNDCLASS` neu gezeichnet wurde. Der Fall, dass dieser Wert `TRUE` wird, wird später besprochen und hat hier noch keine Bedeutung.
- `rcPaint` ist eine weitere Struktur vom Typ `RECT`. Sie enthält die Daten des ungültigen Rechtecks.
- Die anderen drei Funktionsparameter werden vom System intern benutzt.

Die neue Struktur `RECT`:

```
typedef struct _RECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

Die einzelnen Variablen der Struktur haben folgende Bedeutungen:

- Alle vier Variablen sind vom Typ LONG. Dies ist ein 32-Bit-signed-Integer. `left` und `top` geben die obere linke Ecke des Rechtecks in Pixeln an. Diese Angaben beziehen sich auf den Abstand vom oberen linken Rand des Gerätekontextes. Die Variablen `right` und `bottom` beziehen sich ebenfalls auf den Abstand zur oberen linken Ecke des Gerätekontexts und geben die untere rechte Ecke des Rechtecks in Pixeln an.

### *Die Funktion SelectObject*

Die Funktion `SelectObject` weist einem Gerätekontext ein Objekt zu. Dieses Objekt ist dann mit dem Gerätekontext verbunden. Alle Funktionen, die den Gerätekontext verwenden, beziehen sich auf Objekte, die mit diesem verbunden sind.

```
HGDIOBJ SelectObject( HDC hdc, HGDIOBJ hgdiobj);
```

Diese Funktion wird im Quelltext der Beispielanwendung zweimal benötigt:

```
SelectObject (hdc, hBrush);  
SelectObject (hdc, hPen);
```

Die Funktionsparameter haben folgende Bedeutung:

- `hdc` verlangt den Handle auf ein Gerätekontext-Objekt.
- `hgdiobj` verlangt den Handle auf ein GDI-Objekt. Dies können z. B. Brush- oder Pen-Objekte sein. Das Gerätekontext-Objekt verweist danach auf die GDI-Objekte, die dem Gerätekontext zugeordnet wurden.
- Als `return` liefert die Funktion den Handle auf das vorherige GDI-Objekt, das mit dem Gerätekontext verknüpft war. Falls die Funktion fehlschlägt, gibt sie NULL zurück.

### *Die einfachen GDI-Zeichenfunktionen*

Als einfache GDI-Zeichenfunktionen bezeichne ich die Funktionen mit wenigen Parametern. Diese Funktionen verwenden auch noch keine Arrays. Sie dienen dazu, grundlegende geometrische Formen zu zeichnen. Die Funktionen, um die es hier geht, sind folgende.

Die Funktion `MoveToEx` zeichnet noch nichts, sondern legt die Position eines Zeichencursors auf dem Gerätekontext fest: also einfach eine Position auf dem Gerätekontext, die von anderen Funktionen benutzt wird. Es folgt die Syntax der Funktion:

```
BOOL MoveToEx( HDC hdc, int X, int Y, LPPOINT lpPoint);
```

Die Funktion wird im Quelltext so verwendet:

```
MoveToEx (hdc, 20, 20, NULL);
```

Die Bedeutung der Funktionsparameter:

- `hdc` legt den Gerätekontext fest, von dem der Wert dieser Position gesetzt werden soll.
- `X, Y` legen die Position fest. Beide Parameter beziehen sich auf die obere linke Ecke. Von dieser Ecke aus wird der Abstand angegeben. Der Pixel in der obersten linken Ecke ist 0,0.
- `lpPoint` erwartet den Zeiger auf eine Struktur vom Typ `POINT`. In dieser Struktur werden dann die Daten der vorherigen Position abgespeichert. (Die Struktur `Point` wurde schon in Kapitel 1 besprochen.)
- Als `return` gibt die Funktion bei einem Fehler 0, sonst ungleich 0 aus.

Die nächste Funktion ist `LineTo`. Sie zeichnet eine Linie mit den Daten des Pen-Objekts in den Gerätekontext. Die Linie beginnt bei der durch die Funktion `MoveTo` festgelegten Position und endet bei den in `LineTo` festgelegten Parametern. Die Syntax der Funktion sieht so aus:

```
BOOL LineTo( HDC hdc, int nXEnd, int nYEnd);
```

Im Quelltext wird die Funktion folgendermaßen verwendet:

```
LineTo (hdc, 100, 100);
```

Nun zu den einzelnen Parametern:

- Der Parameter `hdc` gibt den Handle auf ein Gerätekontext an.
- `nXEnd, nYEnd` geben die Position im Gerätekontext von der oberen linken Ecke aus an.

Eine weitere Funktion ist `Rectangle`. Sie zeichnet ein Rechteck. Das Rechteck ist mit den im Pen-Objekt angegebenen Eigenschaften umrandet und mit den in dem Brush-Objekt angegebenen Eigenschaften ausgefüllt. Es folgt die Funktionssyntax.

```
BOOL Rectangle( HDC hdc, int nLeftRect, int nTopRect,  
               int nRightRect, int nBottomRect);
```

Im Quelltext wird die Funktion auf folgende Weise verwendet.

```
Rectangle (hdc, 120, 20, 240, 140);
```

Die Parameter der Funktion:

- `hdc` ist der Handle auf einen Gerätekontext.
- `nLeftRect`, `nTopRect`, `nRightRect`, `nBottomRect` geben die Position des Rechtecks in dem Gerätekontext an. Besser gesagt: die Position des Rechtecks im Speicher des Bildschirms, auf den das Gerätekontext-Objekt verweist. Der Gerätekontext ist also der Speicher eines Gerätes, auf welches das Gerätekontext-Objekt verweist. Alle Werte werden in Pixeln angegeben.

Die nächste Funktion ist `RoundRect`. Sie ist fast identisch mit `Rectangle`, besitzt aber zwei Parameter mehr. Sie füllt ein Rechteck mit abgerundeten Ecken mit den Eigenschaften im Brush-Objekt aus. Der Rand wird mit den Eigenschaften im Pen-Objekt gezeichnet.

```
BOOL RoundRect( HDC hdc, int nLeftRect, int nTopRect,  
               int nRightRect, int nBottomRect,  
               int nWidth, int nHeight);
```

Im Quelltext sieht diese Funktion so aus:

```
RoundRect (hdc, 260, 20, 420, 140, 20, 20);
```

Die einzelnen Parameter :

- `hdc` gibt den Handle auf das Gerätekontext-Objekt an.
- `nLeftRect`, `nTopRect`, `nRightRect`, `nBottomRect` geben wie bei `Rectangle` die Position des Rechtecks in Pixeln an.
- `nWidth`, `nHeight` setzen die Breite und Höhe der Ellipse fest, mit der die Ecken des Rechteckes abgerundet werden.

Die nächste Funktion zeichnet nicht in einen Gerätekontext, also einen Speicherbereich auf dem Bildschirm, sondern bezieht sich auf die Struktur `RECT`, die schon besprochen wurde. Sie füllt die Variablen der Struktur `RECT` mit den angegebenen Parametern aus, so dass man sie nicht selbst zuweisen muss.

```
BOOL SetRect( LPRECT lprc, int xLeft, int yTop,
              int xRight, int yBottom);
```

Die Funktion wird im Quelltext wie folgt verwendet.

```
SetRect (&rect, 20, 260, 240, 420);
```

Ihre Parameter sind die folgenden:

- `lprc` verlangt den Zeiger auf eine `RECT`-Struktur.
- Dieser `RECT` Struktur werden die folgenden vier Parameter zugeordnet.

Die Funktion `FrameRect` nutzt eine Struktur des Typs `RECT`, um ein Rechteck ohne Füllung und nur mit Rand zu zeichnen. Die Eigenschaften des Randes werden über ein `Brush`-Objekt angegeben.

```
int FrameRect( HDC hdc, CONST RECT *lprc, HBRUSH hbr);
```

Die Funktion sieht im Quelltext so aus:

```
FrameRect (hdc, &rect, hBrush);
```

- `hdc` gibt den Handle auf das Gerätekontext-Objekt an.
- `lprc` verlangt einen Zeiger auf eine `RECT`-Struktur.
- `hbr` gibt ein separates `Brush`-Objekt an, das diese Funktion zum Zeichnen verwendet. Es wird nicht das `Brush`-Objekt des Gerätekontext-Objekts benutzt.

Die nächste Funktion heißt `FillRect`. Sie hat die gleichen Parameter wie `FrameRect` und funktioniert auf die gleiche Art und Weise. Der Unterschied liegt darin, dass diese Funktion das angegebene Rechteck ausfüllt.

```
int FillRect( HDC hdc, CONST RECT *lprc, HBRUSH hbr);
```

Die Funktion wird im Quelltext folgendermaßen verwendet:

```
FillRect (hdc, &rect, hBrush);
```

Und die Parameter haben folgende Bedeutungen:

- `hDC` ist der Handle auf das Gerätekontext-Objekt
- `lprc` ist ein Zeiger auf eine RECT-Struktur.
- `hbr` ist der Handle auf ein Brush-Objekt, das diese Funktion zum Zeichnen verwenden soll.

Eine weitere und letzte Funktion, die in diesem Buch vorgestellt wird, ist `Ellipse`. Sie zeichnet eine Ellipse auf den Gerätekontext. Der Rand der Ellipse wird vom Pen-Objekt und die Ausfüllung des Objekts vom Brush-Objekt des Gerätekontext-Objekts angegeben.

```
BOOL Ellipse( HDC hdc, int nLeftRect, int nTopRect,  
             int nRightRect, int nBottomRect);
```

Im Quelltext wird die Funktion wie folgt verwendet.

```
Ellipse (hdc, 440, 260, 480, 420);
```

Die Parameter der Funktion sehen folgendermaßen aus:

- `hdc` gibt den Handle auf das Gerätekontext-Objekt an.
- `nLeftRect`, `nTopRect`, `nRightRect` und `nBottomRect` geben einen rechteckigen Bereich des Gerätekontexts an, in dem die Ellipse gezeichnet wird.

### *Pen- und Brush-Objekte wieder entfernen*

Nachdem mit den erstellten Pen- und Brush-Objekten gezeichnet wurde, sollen diese wieder aus dem Speicher entfernt werden. Dann sollen die alten Pen- und Brush-Objekte wiederhergestellt werden.

```
DeleteObject (hPen);  
DeleteObject (hBrush);
```

Dies ist erforderlich, wenn die Objekte immer wieder erstellt werden. Soll ein Gerätekontext mit einem Objekt verbunden bleiben, muss dieses natürlich nicht entfernt werden.

## *EndPoint*

Die Funktion `EndPaint` setzt das Gerätekontext-Objekt wieder frei.

```
EndPaint (hWnd, &ps);
```

Ein Gerätekontext sollte immer gleich nach dem Zeichnen wieder freigesetzt werden.



# 3 Textausgabe mit der GDI

## 3.1 Allgemeines

In diesem Kapitel dreht sich alles um die Textausgabe mit der GDI. Dazu muss erst einmal Grundlegendes gesagt werden. Die Tastatur liefert Scan-Codes zurück. Diese Scan-Codes bezeichnen jede Taste eindeutig. Das Problem besteht nun darin, diesen Scan-Codes einen weiteren Code zuzuordnen, der landesspezifisch ist und ein Zeichen angibt. Dafür gibt es Zeichensätze, die Werten Zeichen zuordnen. Diese Codes variieren teilweise landesspezifisch, der Standard ist der ISO-Standard. Er umfasst die Codes von 0x20 bis 0x7F. Der ASCII-Code schließt diesen ISO-Standard mit ein und definiert die unteren Werte von 0x00 bis 0x19. Der erweiterte ASCII-Code definiert auch noch die oberen 128 Werte. Diese Werte werden mit Blockgrafiken belegt. Den erweiterten ASCII-Code nennt man auch den OEM-Zeichensatz. Windows verwendet eigentlich keinen dieser Zeichensätze, sondern es wurde eigens für Windows ein neuer Standard definiert. Dieser neue Zeichensatz heißt ANSI. Er beinhaltet den ISO-Standard-Zeichensatz, also die Definition von 0x20 bis 0x7F. Von ANSI gibt es mehrere Varianten, die sich auf die Veränderungen oberhalb von 0x7F bis 0xFF beziehen. Die Codes von 0x00 bis 0x1F sind nicht definiert. Die westeuropäische ANSI-Version trägt auch den Namen ISO-8859. Compiler wie C und C++ benötigen den Quelltext, der auf der Grundlage dieses Zeichensatzes beruht. Das heißt man muss die Funktionen mit diesem Zeichensatz schreiben.

Es gibt zwei verschiedene Arten von Schriften unter Windows: die GDI-Schriften und die True-Type-Schriften. Die GDI-Schriften sind Bitmap-Schriften. Es wird einfach ein Raster mit gesetztem Bit und nicht gesetztem Bit auf das Ausgabemedium übertragen. Diese Schriften kann man nur schlecht in ihrer Größe verändern, aber gerade bei Grafikanwendungen sehr schnell einsetzen. Sie können mehrere landesspezifische Zeichensätze enthalten und sind in Da-

teien mit der Endung FONT abgespeichert. Es gibt dort proportionale und nicht-proportionale Schriften. True-Type-Schriftarten dagegen sind Vektor-Schriftarten. Sie enthalten Koordinaten von Pixeln, die durch Linien verbunden werden. Sie sind deshalb sehr gut dazu geeignet, in verschiedenen Größen dargestellt zu werden. Da sie aber jedes Mal neu berechnet werden müssen, sind sie für die schnelle Grafikausgabe weniger gut geeignet. Auch die True-Type-Schriftarten enthalten mehrere landesspezifische Zeichensätze.

In diesem Kapitel geht es darum, wie man die Bitmap-Schriftarten verwendet.

## 3.2 Text mit der GDI erstellen

### 3.2.1 Der Quelltext

Die nächste Anwendung ist ein Beispiel für die Verwendung von Schriftarten. Es werden allerdings noch keine True-Type-Schriftarten verwendet, sondern nur verschiedene Standard-Windowszeichensätze, also Schriftarten. Die Anwendung nach dem Start sehen Sie in Abbildung 3.1.



Bild 3.1: Schriftarten nach dem Start

```

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWnd;
    hWnd = CreateWindow("WinProg","Fenster",
                      WS_OVERLAPPEDWINDOW,
                      0,0,600,460,NULL,NULL,
                      hInstance, NULL);

    ShowWindow (hWnd, nCmdShow);

    UpdateWindow (hWnd);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
    {
        DispatchMessage(&Message);
    }
}

```

```
        return (Message.wParam);
    }

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_PAINT:
            HDC hdc;
            PAINTSTRUCT ps;
            hdc = BeginPaint (hWnd, &ps);

            HFONT hFont;

            hFont = (HFONT) GetStockObject ( SYSTEM_FONT);
            SelectObject (hdc, hFont);
            SetTextColor (hdc, RGB(0,0,180));
            SetBkColor (hdc, RGB(190,180,200));
            SetTextAlign (hdc, TA_LEFT);
            char *string1;
            string1 = new char[12];
            lstrcpy (string1,"SYSTEM_FONT");
            TextOut (hdc, 10, 10, string1, lstrlen(string1));

            hFont =
                (HFONT) GetStockObject (SYSTEM_FIXED_FONT);
            SelectObject (hdc, hFont);
            delete (string1);
            string1 = new char[80];
            lstrcpy (string1,"SYSTEM_FIXED_FONT");
            TextOut (hdc, 10, 40, string1, lstrlen(string1));

            hFont = (HFONT) GetStockObject (ANSI_FIXED_FONT);
            SelectObject (hdc, hFont);
            delete (string1);
            string1 = new char[80];
```

```

lstrcpy (string1,"ANSI_FIXED_FONT");
TextOut (hdc, 10, 80, string1, lstrlen(string1));

hFont = (HFONT) GetStockObject ( ANSI_VAR_FONT);
SelectObject (hdc, hFont);
delete (string1);
string1 = new char[80];
lstrcpy (string1,"ANSI_VAR_FONT");
TextOut (hdc, 10, 120, string1,
        lstrlen(string1));

hFont =
    (HFONT) GetStockObject ( DEFAULT_GUI_FONT);
SelectObject (hdc, hFont);
delete (string1);
string1 = new char[80];
lstrcpy (string1,"DEFAULT_GUI_FONT");
TextOut (hdc, 10, 160, string1,
        lstrlen(string1));

hFont = (HFONT) GetStockObject ( OEM_FIXED_FONT);
SelectObject (hdc, hFont);
delete (string1);
string1 = new char[80];
lstrcpy (string1,"OEM_FIXED_FONT");
TextOut (hdc, 10, 200, string1,
        lstrlen(string1));

EndPaint (hWnd, &ps);
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                        wParam, lParam);
}
}

```



### 3.2.2 Die Bedeutung des Quelltexts

Der Quelltext baut auf dem Grundgerüst der ersten Anwendung auf. Die Abweichungen beziehen sich im Gegensatz zum zweiten Quelltext nur auf den Bereich WM\_PAINT.

#### Ein neuer Handle

Die erste Veränderung im Quelltext findet sich im Wort HFONT. Dieses Wort deklariert eine Variable vom Typ HFONT. Dies ist ein Handle auf einen Font, also eine Schriftart bzw. einen Zeichensatz. Auch die Schriftarten sind Objekte.

#### Den Handle auf eine vordefinierte Schriftart erhalten

Mit der Funktion `GetStockObject` kann man den Handle auf eine vordefinierte Schriftart erhalten. Sie müssen diese Schriftart durch eine Konstante angeben. Dann gibt die Funktion einen Handle auf das Schriftarten-Objekt zurück. Da diese Funktion mehr kann als das, müssen Sie einen allgemeinen GDI-Handle-Wert in einen Handle für eine Schriftart umwandeln, also einen Font. Die Funktionssyntax folgt jetzt.

```
HGDIOBJ GetStockObject( int fnObject);
```

Die Funktion taucht insgesamt sechsmal auf. Jedesmal holt sie den Handle für einen anderen Font.

```
hFont = (HFONT) GetStockObject ( SYSTEM_FONT);  
hFont = (HFONT) GetStockObject ( SYSTEM_FIXED_FONT);  
hFont = (HFONT) GetStockObject ( ANSI_FIXED_FONT);  
hFont = (HFONT) GetStockObject ( ANSI_VAR_FONT);  
hFont = (HFONT) GetStockObject ( DEFAULT_GUI_FONT);  
hFont = (HFONT) GetStockObject ( OEM_FIXED_FONT);
```

Der Funktionsparameter hat die folgende Bedeutung.

- `fnObject` gibt eine Konstante an. Je nach Konstante wird ein Handle auf ein anderes Objekt zurückgegeben.
- Als `return` wird der Handle auf ein Objekt zurückgeliefert, das durch die Konstante spezifiziert wurde. Dieser Handle ist aber von einem allgemeinen Typ. Er muss vorher noch umgewandelt werden.

## Die Schriftart dem Gerätekontext zuweisen

Der Handle des Schriftart-Objekts wird dem Gerätekontext-Objekt zugewiesen. Das heißt es wird ein Verweis vom Gerätekontext-Objekt auf das Schriftart-Objekt gesetzt. Zu diesem Zweck wird die Funktion `SelectObject` verwendet. Diese Funktion steht im Zusammenhang mit vielen GDI-Objekten. Man übergibt der Funktion den passenden Handle auf das Objekt, und sie macht alles Weitere. Da die Funktion schon in Kapitel 2 besprochen wurde, gehen wir hier nicht mehr im Einzelnen auf sie ein.

## Die Schriftfarbe

Das Gerätekontext-Objekt speichert zwei Eigenschaften für die Farbe der Schrift. Die beiden Werte des Gerätekontext-Objekts bestimmt man mit den Funktionen `SetTextColor` und `SetBkColor`. Der Wert, den man mit `SetTextColor` festlegt, wird bei der Textausgabefunktion `TextOut` als Vordergrundfarbe benutzt. Der Wert wiederum, den man mit `SetBkColor` setzt, wird bei der `TextOut`-Funktion als Hintergrundfarbe verwendet. Die Funktion `TextOut` wird gleich besprochen. Als Nächstes folgt die Syntax der Funktionen `SetTextColor` und `SetBkColor`.

```
COLORREF SetTextColor(HDC hdc, COLORREF crColor);  
COLORREF SetBkColor(HDC hdc, COLORREF crColor);
```

Im Quelltext tauchen diese Funktionen wie folgt auf:

```
SetTextColor (hdc, RGB(0,0,180));  
SetBkColor (hdc, RGB(190,180,200));
```

Als Erstes betrachten wir die Parameter der Funktion `SetTextColor`.

- `hdc` ist wie immer der Handle auf ein Gerätekontext-Objekt. Bei diesem Gerätekontext-Objekt werden die Eigenschaften gesetzt.
- `crColor` ist eine Variable vom Typ `COLORREF`. Dieser Typ wurde schon in Kapitel 2 besprochen. Er gibt eine RGB-Farbe an. Diese RGB-Farbe wird als Eigenschaft in den Gerätekontext eingesetzt. Diese Eigenschaft wird, wenn die Funktion `SetTextColor` aufgerufen wird, von der Funktion `TextOut` für die Vordergrundfarbe benutzt.

Nun zu den Funktionsparametern der Funktion `SetBkColor`:

- `hdc` ist wie immer der Handle auf ein Gerätekontext-Objekt. Bei diesem Gerätekontext-Objekt werden die Eigenschaften gesetzt.
- `crColor` ist eine Variable vom Typ `COLORREF`. Dieser Typ wurde schon in Kapitel 2 besprochen. Er gibt eine RGB-Farbe an. Diese RGB-Farbe wird als Eigenschaft in den Gerätekontext eingesetzt. Diese Eigenschaft wird, falls die Funktion `SetBkColor` aufgerufen wird, von der Funktion `TextOut` für die Hintergrundfarbe benutzt.

## Die Textausrichtung festlegen

Zur Bestimmung der Textausrichtung wird wieder die Eigenschaft eines Gerätekontext-Objekts gesetzt. Diese Eigenschaft wird mit der Funktion `SetTextAlign` festgelegt. Natürlich können alle gesetzten Eigenschaften eines Gerätekontext-Objekts von Funktionen unterschiedlich ausgelegt werden. Daher benutzt auch die Funktion `TextOut` diese Eigenschaften individuell. Das bedeutet, wenn hier `TA_CENTER` als Konstante angegeben wurde, benutzt `TextOut` diesen Wert, um den Text um die angegebene Position herum zu positionieren. Dagegen benutzt die Funktion `DrawText` diesen Wert dazu, um in einem angegebenen Rechteck die Mitte zu finden. Nun zur Syntax der Funktion `SetTextAlign`:

```
UINT SetTextAlign( HDC hdc, UINT fMode);
```

Im Quelltext heißt die Funktion so:

```
SetTextAlign (hdc, TA_LEFT);
```

Die Funktionsparameter:

- `hdc` ist ein Handle auf ein Gerätekontext-Objekt.
- `fMode` ist ein Wert, der mit Konstanten bestimmt wird.

## Die Windows-String-Funktionen

Sie haben sich wahrscheinlich schon über die Verwendung der Funktion `lstrcpy` anstatt `strcpy` gewundert. Windows hat seine eigenen String-Funktionen. Das heißt man benötigt keine anderen Bibliotheken zur Verarbeitung von Strings. Zu diesen Funktionen gehören un-



ter anderem `lstrcpy`, `lstrcat`, `lstrcmp` und `lstrlen`. Es wird einfach ein `l` voran gesetzt. Die Syntax verändert sich nicht.

```
LPTSTR lstrcpy( LPTSTR lpString1, LPCTSTR lpString2);
```

Diese Funktion wird im Quelltext wie folgt verwendet:

```
lstrcpy (string1,"SYSTEM_FONT");
lstrcpy (string1,"SYSTEM_FIXED_FONT");
lstrcpy (string1,"ANSI_FIXED_FONT");
lstrcpy (string1,"ANSI_VAR_FONT");
lstrcpy (string1,"DEFAULT_GUI_FONT");
lstrcpy (string1,"OEM_FIXED_FONT");
```

Die Funktionsparameter haben folgende Bedeutung:

- `lpString1` ist vom Typ `LPTSTR`. Dies ist ein Zeiger auf ein `Char-Array`. Hier wird also der String verlangt, an dessen Stelle der andere String treten soll.
- `lpString2` ist dieser andere String. Er ist vom Typ `LPCTSTR`, was bedeutet, dass damit nur das Wort `const` zum obigen Typ `LPTSTR` ergänzt wurde.

## Die Textausgabe

Zur Textausgabe wird die Funktion `TextOut` verwendet. Vor allem die Turbo-Pascal-Programmierer werden hier Probleme haben, denn dort ist die Funktion der Unitgraph `OutText`. Diese Textfunktion bezieht sich wie alle Grafikfunktionen auf einen Handle auf ein Gerätekontext-Objekt. Sie verwendet die Werte, die im Gerätekontext-Objekt durch die Funktionen `SetTextColor` für die Vordergrundfarbe, `SetBkColor` für die Hintergrundfarbe und die Funktion `SetTextAlign` für die Ausrichtung des Texts festgelegt werden. Es folgt die Syntax:

```
BOOL TextOut( HDC hdc, int nXStart, int nYStart,
             LPCTSTR lpString, int cbString);
```

Die Funktion wurde folgendermaßen verwendet:

```
TextOut (hdc, 10, 10, string1, lstrlen(string1));
TextOut (hdc, 10, 40, string1, lstrlen(string1));
TextOut (hdc, 10, 80, string1, lstrlen(string1));
TextOut (hdc, 10, 120, string1, lstrlen(string1));
```

```
TextOut(hdc, 10, 160, string1, lstrlen(string1));
TextOut(hdc, 10, 200, string1, lstrlen(string1));
```

Die Funktionsparameter haben folgende Bedeutungen.

- `hdc` ist der Handle auf das Gerätekontext-Objekt.
- `nXStart`, `nYStart` sind vom Variablentyp `int`. Sie geben eine Position an, die sich auf `SetTextAlign` bezieht. Diese Position wird in Pixeln von oben links angegeben. Falls `SetTextAlign` z. B. die Konstante `TA_LEFT` benutzt, geben `nXStart` und `nYStart` die obere linke Ecke des Texts an. Falls `TA_CENTER` angegeben worden ist, geben die beiden Werte die obere Position in der Mitte an. Der Text baut sich also um den angegebenen Punkt herum auf.
- `lpString` ist wieder ein Zeiger auf einen `char`-Array, also ein String. Dies ist der Text, der ausgegeben wird.
- `cbString` ist vom Variablentyp `int`. Dieser Wert gibt die Anzahl der Zeichen an, die angezeigt werden sollen. Um den ganzen String anzuzeigen, also um die Länge des Strings zu ermitteln, verwendet man `lstrlen`.

Die Funktion `lstrlen` sollte auch bekannt sein. Mit ihr kann man die Länge eines Strings ermitteln.

```
int lstrlen( LPCTSTR lpString)
```

Sie taucht im Quelltext nur in Zusammenhang mit der Funktion `TextOut` auf und es gibt nur einen Funktionsparameter.

- `lpString` ist der String.
- Als `return` erhält man die Anzahl der Zeichen. Dabei wird die abschließende `NULL` des Strings nicht als Zeichen gezählt.

# 4 Steuerelemente

## 4.1 Allgemeines

Unter Steuerelemente versteht man Buttons, Edit-Felder oder ListBoxen. Sie regeln die Kommunikation mit dem Benutzer. Steuerelemente sind, wie alles andere auf dem Bildschirm auch, Fenster. Das bedeutet: Steuerelemente werden mit der Funktion `CreateWindow` angelegt. Für einen Button wird demzufolge die Fensterklasse `BUTTON` angegeben. Diese Fensterklasse wird genauso definiert, wie alle anderen auch. Die Windows-Steuerelemente besitzen also keine Sonderstellung und haben eine Funktion zur Nachrichtenverarbeitung. Wenn das Steuerelement die Nachricht `WM_PAINT` erhält, zeichnet es einen Button in sein Fenster. Es reagiert auch auf Mausclicks. Erhält das Steuerelement einen Mausclick auf den Button, sendet es sich selber die Nachricht `BN_CLICKED`. Bei den Steuerelementen gibt man normalerweise ein Fenster an, das in der Hierarchie der Fenster über den anderen steht. Dies hat normalerweise nur Einflüsse auf die Grafik, wie die Positionierung des Fensters und die grafische Anzeige nur innerhalb des Fensters höherer Ordnung. Windows-Steuerelemente reagieren ebenfalls auf diese Hierarchie. Sie ermitteln das Fenster höherer Ordnung und geben bei einem Klick auf dem Button noch die Nachricht `WM_COMMAND` an dieses Fenster ab. Da die Windows-Steuerelemente eine Nachricht an die Fenster höherer Ordnung senden, bekommen diese eigenständigen Fenster den Anschein, Attribute des Fensters höherer Ordnung zu sein. Das sind sie aber nicht. Das wichtigste ist, sich zu merken, dass Steuerelemente ganz normale Fensterobjekte sind, die zur Nachrichtenverarbeitung auf eine Funktion zugreifen, über die man keine Informationen hat. So können diese Windows-Standardsteuerelemente sehr gut dazu verwendet werden Benutzereingaben aufzufangen.

## 4.2 Eine Anwendung mit einem Steuerelement

In diesem Kapitel geht es darum, ein Steuerelement, nämlich einen Button, zu erzeugen. Der Button sendet eine Nachricht an das Fenster höherer Ordnung, die von der Beispielanwendung verarbeitet wird. Die Anwendung gibt alle Zeichen des momentanen `SYSTEM_FONTs` aus (länderspezifischer ANSI). Diesen Zeichen wird, je nach Ländereinstellung, die Ausgabe des Tastaturtreibers angepasst. Der Scan-Code der Taste wird vom Tastaturtreiber ausgewertet. Wenn Sie das deutsche Tastaturlayout gewählt haben, werden den Scan-Codes entsprechend die Werte des deutschen ANSI-Zeichensatzes zugeordnet. Der deutsche ANSI-Zeichensatz und das deutsche Tastaturlayout (bzw. der Tastaturtreiber) werden bei der Installation von Windows eindeutig festgelegt. Jedem ANSI-Zeichensatz liegt aber der ISO-Zeichensatz zugrunde. Auch gilt für mehrere Länder ein ANSI-Zeichensatz. Auch True-Type-Schriftarten sind ANSI-Zeichensätze, allerdings für verschiedene Länder. Der Unterschied zu System-Zeichensätzen wie z.B. dem `SYSTEM_FONT` zeigt sich in der Definition der Zeichen. Die System-Schriften sind wie Bilder durch Pixel definiert. Die True-Type-Schriftarten werden durch Vektoren und weitere Zusätze festgelegt und müssen erst berechnet werden. Die System-Zeichensätze werden Pixel für Pixel schnell kopiert.

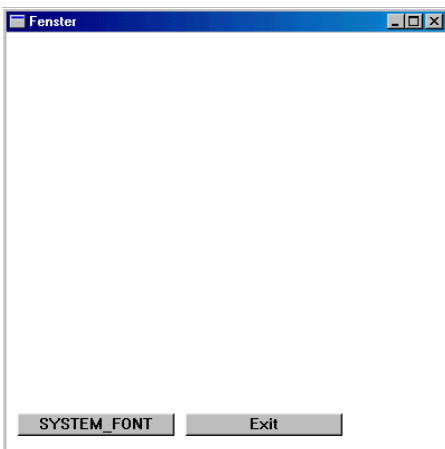


Bild 4.1: Die Anwendung nach dem Start



```
WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hWnd;
hWnd = CreateWindow("WinProg","Fenster",
                   WS_OVERLAPPEDWINDOW,
                   0,0,400,400,NULL,NULL,
                   hInstance, NULL);

ShowWindow (hWnd, nCmdShow);

UpdateWindow (hWnd);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message)
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_CREATE:
            HWND hButton, hButtonExit;
            hButton = CreateWindow("BUTTON","SYSTEM_FONT",
                                  WS_CHILD | WS_VISIBLE |
                                  BS_PUSHBUTTON,
                                  10,340,140,20,
                                  hWnd,(HMENU) 1,
```

```

        hInstGlobal, NULL);
hButtonExit = CreateWindow("BUTTON","Exit",
        WS_CHILD |
        WS_VISIBLE |
        BS_PUSHBUTTON,
        160,340,140,20,
        hWnd,(HMENU) 2,
        hInstGlobal, NULL);

return 0;
case WM_COMMAND:
    if (HIWORD(wParam) == BN_CLICKED)
    {
        if (LOWORD(wParam) == 1)
        {
            HDC hdc;
            hdc = GetDC (GetParent((HWND) lParam));
            HFONT hFont;
            hFont = (HFONT)
                GetStockObject(SYSTEM_FONT);
            SelectObject (hdc, hFont);
            TEXTMETRIC tm;
            GetTextMetrics (hdc, &tm);
            int i;
            int ix,iy;
            ix = 0;
            iy = 0;
            char *string;
            string = new char[2];
            string[1] = 0;
            for (i=0;i<=255;i++)
            {
                string[0] = i;
                TextOut (hdc, ix, iy, string,
                    strlen(string));
                iy = iy + tm.tmHeight;
                if ((iy/tm.tmHeight) == 16)
                {
                    iy = 0;

```

```

        ix = ix + 20;
    }
}
ReleaseDC (GetParent((HWND) lParam), hdc);
delete [] string;
}
if (LOWORD(wParam) == 2)
{
    SendMessage (GetParent((HWND)lParam),
        WM_DESTROY ,0 ,0);
}
}
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
        wParam, lParam);
}
}

```

#### 4.2.1 Erläuterung des Quelltexts

Der Quelltext ähnelt wieder dem in Kapitel 2. Deshalb gehen wir hier nur auf die wichtigsten Änderungen ein.

##### Ein Steuerelement wird erstellt

Die wichtigste Änderung findet sich in der Fensterfunktion des Fensterobjekts `hWindow`. Dort wird die Message `WM_CREATE` bearbeitet. Diese Message wird bei der Erstellung des Fensterobjekts in die Warteschlange eingetragen. Dann werden weitere Fensterobjekte erstellt. In diesem Fall handelt es sich dabei um Steuerelemente. Mit der Funktion `CreateWindow` wird ein Fenster der Klasse `BUTTON` erstellt. Es ist ein Fenster unterer Ordnung zum Hauptfenster `hWindow`. Dies wird durch den Wert von `hWindow` als Parameter von `CreateWindow` angezeigt und durch `WS_STYLE` bestimmt. Über den Wert 1 wird ein Identifizierungscode an das Fensterobjekt angehängt. Wenn ich in diesem Abschnitt von einem Fenster spreche, meine ich das



Fensterobjekt. Als Fenster wird sonst nur der Bereich des Bildschirms bezeichnet, auf den sich das Fensterobjekt bezieht. Nun zur Funktion `CreateWindow`.

```
HWND CreateWindow(LPCTSTR lpClassName,  
                 LPCTSTR lpWindowName,  
                 DWORD dwStyle,  
                 int x,  
                 int y,  
                 int nWidth,  
                 int nHeight,  
                 HWND hWndParent,  
                 HMENU hMenu,  
                 HANDLE hInstance,  
                 LPVOID lpParam  
                 );
```

Die Funktion wird bei der Erstellung von Steuerelementen zweimal benötigt.

```
hButton = CreateWindow("BUTTON", "SYSTEM_FONT",  
                      WS_CHILD | WS_VISIBLE |  
                      BS_PUSHBUTTON,  
                      10, 340, 140, 20,  
                      hWnd, (HMENU) 1,  
                      hInstGlobal, NULL);  
hButtonExit = CreateWindow("BUTTON", "Exit",  
                           WS_CHILD |  
                           WS_VISIBLE |  
                           BS_PUSHBUTTON,  
                           160, 340, 140, 20,  
                           hWnd, (HMENU) 2,  
                           hInstGlobal, NULL);
```

Hier die Parameter, die von Gewicht sind:

- `lpClassName` ist immer noch der Name der Fensterklasse. Diesmal wird hier die Fensterklasse vom Typ `BUTTON` verwendet. Die Funktion, auf die sich diese Fensterklasse bezieht, ist ebenfalls vordefiniert. Es wird ein ganz normales Fensterobjekt erzeugt.

- `lpWindowName` dient in diesem Fall nicht dazu, den Titel anzugeben. Die Funktion des Fensterobjekts verwendet diesen Wert für die Buttonbeschriftung. Genauso wie man mit dem Attribut `WS_POPUP` und dem Wert von dem Parameter `lpWindowName` ein Fensterobjekt erzeugen kann. Wenn das Attribut `WS_POPUP` anstelle von `WS_OVERLAPPEDWINDOW` steht, interpretiert die Funktion `DefWindowProc` aus der Angabe `WS_POPUP`, dass ein Fenster ohne Zusätze verlangt wurde. Man kann dann den übergebenden String abfragen, und ihn über GDI-Funktionen einsetzen.
- `dwStyle` verändert sich auch. Es muss nun zusätzlich die Konstante `WS_CHILD` angegeben werden. Darauf reagieren die anderen Windows-Funktionen im grafischen Sinne. Denn jetzt wird dieses Fenster nur noch angezeigt, wenn das Fenster höherer Ordnung auch angezeigt wird. Man wollte durch diese Hierarchie eine Struktur für die Zusammengehörigkeit der Anzeige schaffen. Es gibt Fenster höherer Ordnung – sie sind nicht vom Typ `WS_CHILD`. Zu diesen Fenstern gehört auch das Desktopfenster, das vom System zuerst erstellt wird.
- Die Koordinaten `x`, `y` beziehen sich immer auf das Fenster höherer Ordnung; bei keinem gesetzten `WS_CHILD` also auf den Screen, da es kein übergeordnetes Fenster gibt.
- In `hwndParent` wird das Fensterobjekt höherer Ordnung angegeben. Ohne `WS_CHILD` muss dieser Parameter `NULL` sein.
- Da Steuerelemente normalerweise kein Menü haben, wird dieser Parameter dafür genutzt, dem Fensterobjekt eine Zahl zuzuordnen, die das Steuerelement dann identifizieren kann.

### Die Nachricht `WM_COMMAND`

Die nächste Änderung liegt in der Behandlung der Nachricht `WM_COMMAND`. Die Fensterobjekte des Typs `BUTTON` senden, wenn auf sie geklickt wird, eine Nachricht an das Fenster höherer Ordnung. Daraufhin kann eine Aktion erfolgen, die man unter `WM_COMMAND` festlegt. Hier soll als Aktion ein Zeichensatz ausgegeben werden, und zwar der ANSI-Zeichensatz der Codepage für Deutschland. Als Codepage bezeichnet man die verschiedenen Zeichensätze mit den

jeweiligen Länderunterschieden. Die Nachricht `WM_COMMAND` hat natürlich mehrere Werte in ihren Übergabeparametern.

```
WM_COMMAND
wNotifyCode = HIWORD(wParam);
wID = LOWORD(wParam);
hwndCtl = (HWND) lParam;
```

`HIWORD` und `LOWORD` sind Makros. Da alle Parameter vom Typ 32-Bit Integer sind, kann man die unteren und oberen 16 Bit einzeln ermitteln. Dies erledigen die Makros `LOWORD` und `HIWORD`.

- `wNotifyCode` ist ein Wert, der durch die Art des Buttons bestimmt wird. Ein Button mit dem Style `BS_PUSHBUTTON` liefert ein `BN_CLICKED` zurück, dabei steht `BN` für *Button Notification*.
- `wID` ist der Identifikationswert der bei `CreateWindow` übergeben wurde.
- `hwndCtl` ist der Handle auf das Fensterobjekt, also das Steuerelement, das die Nachricht sendet.

#### *Der erste Teil von WM\_COMMAND*

```
HDC hdc;
hdc = GetDC (GetParent((HWND) lParam));
HFONT hFont;
hFont = (HFONT)
    GetStockObject(SYSTEM_FONT);
SelectObject (hdc, hFont);
```

Im ersten Teil von `WM_COMMAND` findet sich nicht viel Neues. Hier wird nur der Handle auf das Gerätekontext-Objekt ermittelt. Außerdem wird der Font `SYSTEM_FONT` dem Gerätekontext-Objekt zugeordnet. Die einzig unbekannt Funktion ist `GetParent`. Mit ihr kann man das Fenster höherer Ordnung ermitteln. Dabei muss der Handle auf das Fensterobjekt angegeben werden, von dem das Fensterobjekt höherer Ordnung ermittelt werden soll.

*Der zweite Teil von WM\_COMMAND*

```
TEXTMETRIC tm;
GetTextMetrics (hdc, &tm);
```

Im zweiten Teil wird die Funktion `GetTextMetrics` benutzt. Sie liefert Informationen über den Zeichensatz zurück. Die Informationen beziehen sich auf den Zeichensatz, der gerade als Objekt mit dem Gerätekontext-Objekt verknüpft ist. Die Informationen werden in einer Struktur mit folgendem Aufbau geliefert.

```
typedef struct tagTEXTMETRIC
{
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    BCHAR tmFirstChar;
    BCHAR tmLastChar;
    BCHAR tmDefaultChar;
    BCHAR tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
} TEXTMETRIC;
```

Betrachten wir hier die wichtigsten Elemente der Struktur:

- `tmHeight` ist vom Typ `LONG` und ein 32-Bit-signed-Integer, wie die meisten anderen auch. Dazu muss man wissen, dass ein Zeichen aus einem `Ascent` und einem `Descent` besteht, was so viel bedeutet wie unterer und oberer Teil. Dies bezieht sich auf eine Grundli-

nie. `Ascent` ist also die Anzahl der Pixel über der Grundlinie der Zeichen und `Descent` die Pixelanzahl darunter. `tmHeight` sind beide Werte zusammengenommen.

- `tmAscent` ist der Wert des `Ascent` der Zeichen.
- `tmDescent` ist der Wert des `Descent` der Zeichen.
- `tmAveCharWeight` ist die durchschnittliche Breite der Zeichen in Pixeln. Natürlich bezieht sich dieser Wert auf den Font, also auf den Zeichensatz.
- `tmMaxCharWeight` ist die maximale Breite der Zeichen in Pixeln im Font.

In diesem Quelltext wird aber nur der Wert der Variablen `tmHeight` verwendet. Jetzt betrachten wir noch einmal die genaue Syntax der Funktion `GetTextMetrics`. Mit ihr füllt man also die Variablen der Struktur `tm` vom Typ `TEXTMETRIC` auf.

```
BOOL GetTextMetrics(HDC hdc, LPTEXTMETRIC lptm);
```

Diese Funktion taucht im Quelltext wie folgt auf.

```
GetTextMetrics (hdc, &tm);
```

Die Funktionsparameter haben folgende Bedeutung:

- `hdc` ist der Handle auf ein Gerätekontext-Objekt von dem die Informationen über das Font-Objekt ermittelt werden sollen, das mit dem Gerätekontext-Objekt verknüpft ist.
- `lptm` ist der Zeiger auf eine Struktur vom Typ `TEXTMETRIC`.

*Der dritte Teil von WM\_COMMAND*

```
int i;  
int ix, iy;  
ix = 0;  
iy = 0;  
char *string;  
string = new char[2];  
string[1] = 0;  
for (i=0; i<=255; i++)  
{  
    string[0] = i;
```

```

TextOut (hdc, ix, iy, string,
        lstrlen(string));
iy = iy + tm.tmHeight;
if ((iy/tm.tmHeight) == 16)
{
    iy = 0;
    ix = ix + 20;
}
}
ReleaseDC (GetParent((HWND) lParam), hdc);
delete [] string

```

Der dritte Teil der Nachricht `WM_COMMAND` ist die Zeichenausgabe auf dem Bildschirm. Natürlich geschieht dies nur im Rahmen des Fensters. Dazu wird ein String abwechselnd mit den Werten von 0 bis 255 gefüllt und ausgegeben. Die Ausgabe erfolgt dabei jeweils in 16 Blöcken. Dafür ist die Schleife verantwortlich. Den Platz, den ein Zeichen vertikal benötigt, wird über `tm.tmHeight` ermittelt. Viel Neues gibt es ansonsten nicht.

#### *Der vierte Teil von WM\_COMMAND*

```

if (LOWORD(wParam) == 2)
{
    SendMessage (GetParent((HWND)lParam),
                WM_DESTROY ,0 ,0);
}
}
return 0;

```

Im letzten Teil der Nachricht wird durch den Klick auf den Button Exit eine Aktion ausgelöst. Hier wird `SendMessage`, eine sehr wichtige Funktion verwendet. Die Funktion sendet eine Nachricht an die entsprechende Anwendung. Dabei gibt es zwei Vorgehensweisen. Wenn die Funktion die Nachricht an dieselbe Anwendung senden soll, von der der Aufruf kommt, wird die Fensterfunktion sofort aufgerufen. Wenn die Nachricht allerdings an eine andere Anwendung gehen soll, wird sie einfach in die Warteschlange dieser anderen Anwendung eingefügt. Dabei tritt folgendes Problem auf: Der Code der sendenden Anwendung wird erst weiter ausgeführt, wenn diese Nachricht

von der anderen Anwendung verarbeitet wurde. Es folgt die Syntax der Funktion `SendMessage`.

```
LRESULT SendMessage(HWND hWnd, UINT Msg,  
                    WPARAM wParam,  
                    LPARAM lParam);
```

Die Funktion taucht im Quelltext so auf:

```
SendMessage (GetParent((HWND)lParam),  
            WM_DESTROY ,0 ,0);
```

Nun zu den Funktionsparametern:

- `hWnd` ist der Handle eines Fensterobjekts, an das die Nachricht gesendet wird.
- `Msg` ist der Wert, der die Nachricht identifiziert.
- `wParam` ist der erste Parameter der Nachricht.
- `lParam` ist der zweite Parameter der Nachricht.

Zum Schluss folgt ein `return`.





# 5 Das Steuerelement Edit-Feld

## 5.1 Allgemeines

Ein weiteres Steuerelement ist das Steuerelement EDIT. Dies ist ein Eingabefeld (Edit-Feld) für Text. Der Benutzer kann hier Text eingeben. Dieses Steuerelement kann genau wie das Steuerelement BUTTON in mehreren Varianten auftreten. In diesem Kapitel gehen wir nur auf die einfache Variante ein.

## 5.2 Der Quelltext

Die Anwendung ist eine Erweiterung der Anwendung aus Kapitel 4. In das Edit-Feld soll ein Name eingegeben werden können. Der Name wird ausgewertet und daraufhin ein Zeichensatz ausgewählt. Alle diese Zeichensätze sind vordefiniert.

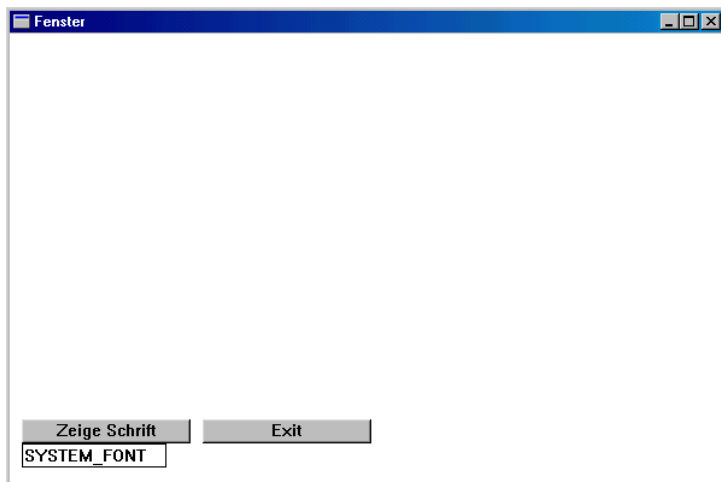


Bild 5.1: Die Anwendung nach dem Start



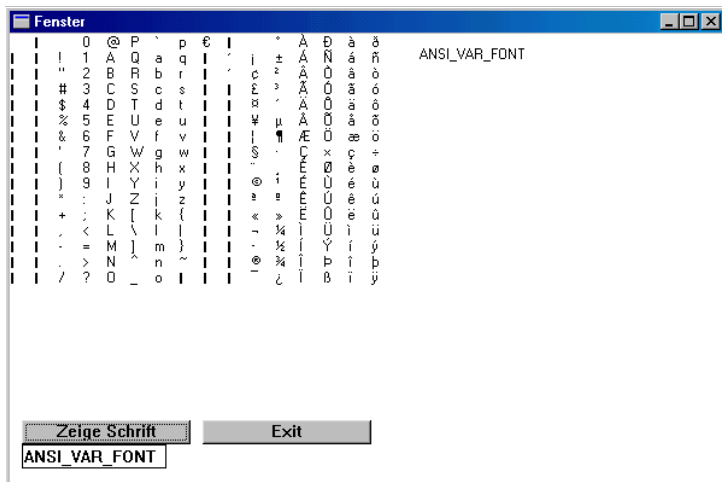


Bild 5.4: Die vordefinierten Font-Objekt ANSI\_VAR\_FONT und DEAFULT\_GUI\_FONT in der Voreinstellung

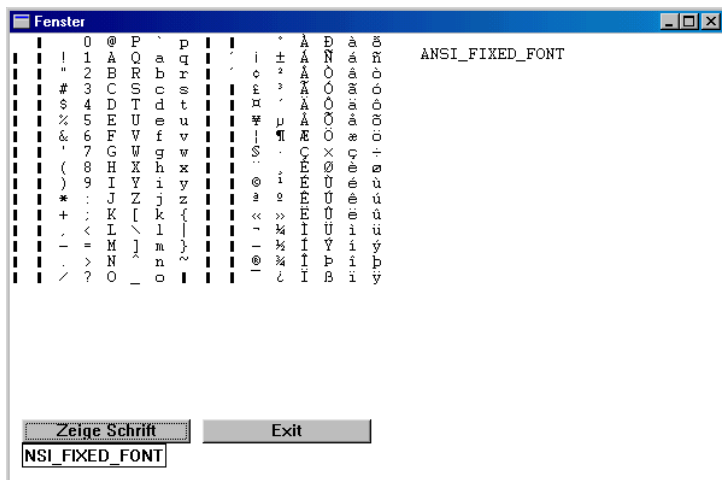


Bild 5.5: Das vordefinierte Font-Objekt ANSI\_FIXED\_FONT



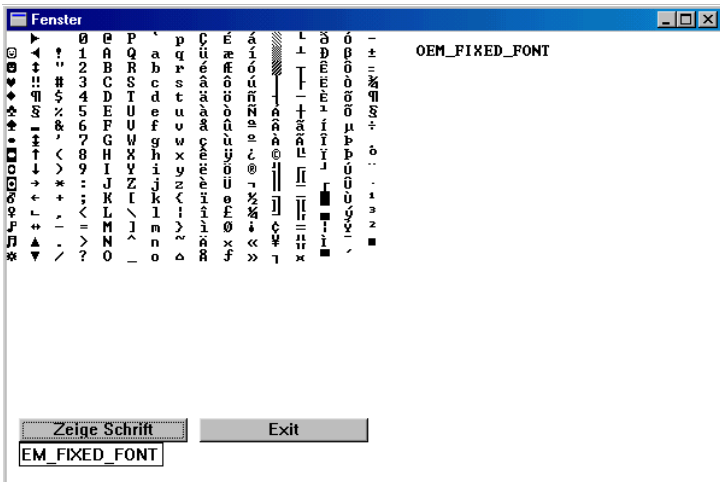


Bild 5.6: Das vordefinierte Font-Objekt OEM\_FIXED\_FONT

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

```
HINSTANCE hInstGlobal;
```

```
HWND hEdit;
```

```
int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
```

```
{
    hInstGlobal = hInstance;
```

```
    WNDCLASS WndClass;
```

```
    WndClass.style = 0;
```

```
    WndClass.cbClsExtra = 0;
```

```
    WndClass.cbWndExtra = 0;
```

```
    WndClass.lpfWndProc = WndProc;
```

```
    WndClass.hInstance = hInstance;
```

```

WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hWnd;
hWnd = CreateWindow("WinProg","Fenster",
                   WS_OVERLAPPEDWINDOW,
                   0,0,600,400,NULL,NULL,
                   hInstance, NULL);

ShowWindow (hWnd, nCmdShow);

UpdateWindow (hWnd);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    TranslateMessage (&Message);
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_CREATE:
            HWND hButton, hButtonExit;
            hButton = CreateWindow("BUTTON","Zeige Schrift",
                                  WS_CHILD | WS_VISIBLE |
                                  BS_PUSHBUTTON,

```



```

        10,320,140,20,
        hWnd,(HMENU) 1,
        hInstGlobal, NULL);
hButtonExit = CreateWindow("BUTTON","Exit",
        WS_CHILD |
        WS_VISIBLE |
        BS_PUSHBUTTON,
        160,320,140,20,
        hWnd,(HMENU) 2,
        hInstGlobal, NULL);

hEdit = CreateWindow("EDIT","SYSTEM_FONT",
        WS_CHILD | WS_VISIBLE |
        WS_BORDER | ES_AUTOHSCROLL,
        10,340,120,20,
        hWnd,(HMENU) 1,
        hInstGlobal, NULL);

return 0;
case WM_COMMAND:
    if (HIWORD(wParam) == BN_CLICKED)
    {
        if (LOWORD(wParam) == 1)
        {
            HDC hdc;
            hdc = GetDC (GetParent((HWND) lParam));
            HFONT hFont;
            char *string1;
            string1 = new char[255];
            SendMessage (hEdit, WM_GETTEXT, 256,
                (LPARAM) string1);

            bool FontDa;
            FontDa = 0;
            if (lstrcmp(string1,"SYSTEM_FONT") == 0)
            {
                hFont = (HFONT)
                    GetStockObject(SYSTEM_FONT);
                FontDa = 1;
            }
            if (lstrcmp(string1,"SYSTEM_FIXED_FONT")

```

```

    ==0)
{
    hFont = (HFONT)
        GetStockObject(SYSTEM_FIXED_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"OEM_FIXED_FONT") == 0)
{
    hFont = (HFONT)
        GetStockObject(OEM_FIXED_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"ANSI_VAR_FONT") == 0)
{
    hFont = (HFONT)
        GetStockObject(ANSI_VAR_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"ANSI_FIXED_FONT")==0)
{
    hFont = (HFONT)
        GetStockObject(ANSI_FIXED_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"DEFAULT_GUI_FONT")
    == 0)
{
    hFont = (HFONT)
        GetStockObject(DEFAULT_GUI_FONT);
    FontDa = 1;
}
if (FontDa == 0)
{
    hFont = (HFONT)
        GetStockObject(SYSTEM_FONT);
    lstrcpy (string1, "SYSTEM_FONT");
}
SelectObject (hdc, hFont);

```



```
RECT rect;
SetRect (&rect, 0, 0, 480, 280);
HBRUSH hBrush;
hBrush = CreateSolidBrush
        (RGB(255,255,255));
FillRect (hdc, &rect, hBrush);
TextOut (hdc, 340, 10, string1,
        lstrlen(string1));
delete [] string1;
TEXTMETRIC tm;
GetTextMetrics (hdc, &tm);
int i;
int ix,iy;
ix = 0;
iy = 0;
char *string;
string = new char[2];
string[1] = 0;
for (i=0;i<=255;i++)
{
    string[0] = i;
    TextOut (hdc, ix, iy, string,
            lstrlen(string));
    iy = iy + tm.tmHeight;
    if ((iy/tm.tmHeight) == 16)
    {
        iy = 0;
        ix = ix + 20;
    }
}
ReleaseDC (GetParent((HWND)lParam), hdc);
delete [] Editstring;
}
if (LOWORD(wParam) == 2)
{
    SendMessage (GetParent((HWND)lParam),
        WM_DESTROY ,0 ,0);
}
```



```

    }
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}

```

### 5.2.1 Was ist neu?

Der Quelltext weist einige Veränderungen auf. Er ist eine Erweiterung des Quelltexts aus Kapitel 4. Über das Edit-Feld kann Text, nämlich der Name der Konstanten, eingegeben werden. Dieser Text-String wird abgefragt und verglichen. Falls der eingegebene Name dem Namen einer Konstanten entspricht, wird das Objekt dieser Konstanten mit dem Gerätekontext-Objekt verbunden.

#### Die erste Neuerung

Zunächst fällt wohl die Erstellung des neuen Steuerelements auf. Es handelt sich um ein Edit-Feld. Dazu benötigen wir die Fensterklasse EDIT. Auch WS\_CHILD muss wieder hinzugefügt werden. Zu WS\_CHILD kommen noch die Konstanten WS\_VISIBLE, WS\_BORDER und ES\_AUTOHSCROLL. WS\_VISIBLE sorgt dafür, dass das Edit-Feld sofort sichtbar ist, WS\_BORDER für den Rand. ES\_AUTOHSCROLL ist ein Style, der nur für Edit-Felder gilt. Er scrollt das Edit-Feld bei der Eingabe von Text, falls der Platz nicht mehr ausreicht.

```

hEdit = CreateWindow("EDIT", "SYSTEM_FONT",
                    WS_CHILD | WS_VISIBLE |
                    WS_BORDER | ES_AUTOHSCROLL,
                    10, 340, 120, 20,
                    hWnd, (HMENU) 1,
                    hInstGlobal, NULL);

```

`hInstGlobal` ist der Handle auf die Anwendung. Dieser muss zunächst einer globalen Variablen übergeben werden.



## Das Ermitteln von Text aus dem Edit-Feld

Man sendet eine Nachricht an das Edit-Feld. Der Text, den das Edit-Feld enthält, ist der Fenstertext. Man ermittelt ihn über die Nachricht `WM_GETTEXT`.

```
SendMessage (hEdit, WM_GETTEXT, 256, (LPARAM) string1);
WM_GETTEXT
wParam = (WPARAM) cchTextMax;
lParam = (LPARAM) lpszText;
```

Die Nachricht `WM_GETTEXT` verlangt mehrere Parameter. In `wParam` muss der Wert der maximalen Zeichenanzahl eingegeben werden, die zurückgeliefert werden soll. In `lParam` muss ein Zeiger einem `char`-Array übergeben werden, der den String enthalten soll.

## Die Auswertung des Textes

Nachdem der Text-String aus dem Edit-Feld eingelesen worden ist, muss er ausgewertet werden. Dazu benutzt man eine ganz normale `if`-Bedingung. Diese Bedingung ist, dass die beiden Strings übereinstimmen. Dabei stammt einer aus dem Edit-Feld und der andere ist eine Abfragekonstante. Danach wird entschieden, auf welches Objekt der Handle zeigen soll. So wird also ein Font ausgewählt. Falls ein Font ausgewählt wurde, wird die Variable `FontDa` auf 1 gesetzt; wurde kein Font ausgewählt, steht die Variable `FontDa` noch auf 0. Dann wird der Font, der mit `SYSTEM_FONT` bezeichnet wird, ausgewählt. Zum Schluss wird dem Gerätekontext-Objekt mit der Funktion `SelectObject` der Handle des eingestellten Fonts zugeordnet.

```
bool FontDa;
FontDa = 0;
if (lstrcmp(string1,"SYSTEM_FONT") == 0)
{
    hFont = (HFONT)
        GetStockObject(SYSTEM_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"SYSTEM_FIXED_FONT")
    ==0)
{
```

```

    hFont = (HFONT)
        GetStockObject(SYSTEM_FIXED_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"OEM_FIXED_FONT") == 0)
{
    hFont = (HFONT)
        GetStockObject(OEM_FIXED_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"ANSI_VAR_FONT") == 0)
{
    hFont = (HFONT)
        GetStockObject(ANSI_VAR_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"ANSI_FIXED_FONT")==0)
{
    hFont = (HFONT)
        GetStockObject(ANSI_FIXED_FONT);
    FontDa = 1;
}
if (lstrcmp(string1,"DEFAULT_GUI_FONT")
    == 0)
{
    hFont = (HFONT)
        GetStockObject(DEFAULT_GUI_FONT);
    FontDa = 1;
}
if (FontDa == 0)
{
    hFont = (HFONT)
        GetStockObject(SYSTEM_FONT);
    lstrcpy (string1, "SYSTEM_FONT");
}
SelectObject (hdc, hFont);

```





Eine Funktion ist neu. Windows stellt auch einen Ersatz für `strcmp` zur Verfügung: er heißt – wie nicht anders zu erwarten – `lstrcmp`. Diese Funktion vergleicht zwei Zeichenketten miteinander. Bei einem Ergebnis, das aussagt, dass beide Strings gleich sind, ist der Rückgabewert 0 ansonsten ungleich 0. Die Funktion vergleicht die Zeichen in der Reihenfolge, in der sie auftauchen und liefert die Differenz der ersten ungleichen Characters zurück. Treten keine auf, ist dieser Wert logischerweise 0.

```
int lstrcmp( LPCTSTR lpString1,
            LPCTSTR lpString2);
```

- `lpString1` ist der erste String für den Vergleich.
- `lpString2` ist der zweite String für den Vergleich.

### Vorbereiten zum Neuschreiben

Nun muss der alte Zeichenbereich darauf vorbereitet werden, neu beschrieben zu werden. Dazu wird ein neues weißes Brush-Objekt erstellt. Mit diesem Objekt wird dann mit der Funktion `FillRect` der Ausgabebereich des Fensters weiß gefärbt. Am Rand kommt noch eine Beschriftung hinzu. Diese Beschriftung enthält den Namen der Konstanten aus der Funktion `GetStockObject`.

```
RECT rect;
SetRect (&rect, 0, 0, 480, 280);
HBRUSH hBrush;
hBrush = CreateSolidBrush
        (RGB(255,255,255));
FillRect (hdc, &rect, hBrush);
TextOut (hdc, 340, 10, string1,
        lstrlen(string1));
```

Die eigentliche Ausgabe entspricht der in Kapitel 4. Daher erfolgt hier keine weitere Erklärung.

# 6 System-Shutdown

## 6.1 Allgemeines

Hinter dem Wort *Shutdown* verbirgt sich das Herunterfahren des Systems. Das bedeutet, das System wird in einen Zustand gebracht, in dem man es abschalten kann. Es gibt unter Windows zwei Wege, ein System herunterzufahren. Der erste besteht darin, das System einfach herunterzufahren, damit der Benutzer den PC ausschalten kann. Die zweite Variante fährt das System herunter und führt gleich danach einen Neustart durch. Anschließend kann man den Benutzer abmelden. Falls der PC das automatische Ausschalten unterstützt, kann sich der PC nach dem Herunterfahren auch noch abschalten.

## 6.2 Eine Anwendung

### 6.2.1 Beschreibung der Anwendung

Die folgende Anwendung soll demonstrieren, wie man die Funktion `ExitWindowsEx` benutzen kann. Das System soll folgende Aktionen ausführen: es soll heruntergefahren und neugestartet werden können, und es soll heruntergefahren werden können mit Ausschalten und der Benutzer soll sich zudem abmelden können. Diese Aktionen sollen einzeln auszuwählen sein. Mit dem Button „Shutdown“ soll die Aktion gestartet werden. Als Zusatz soll es noch die Option „keine Nachricht an laufende Anwendungen“ geben. Sie bewirkt im aktivierten Zustand, dass keine Nachrichten an die laufenden Anwendungen geschickt werden, bevor sie beendet werden. Normalerweise werden nämlich Nachrichten an laufende Anwendungen geschickt, damit diese noch die Möglichkeit haben Daten zu sichern.

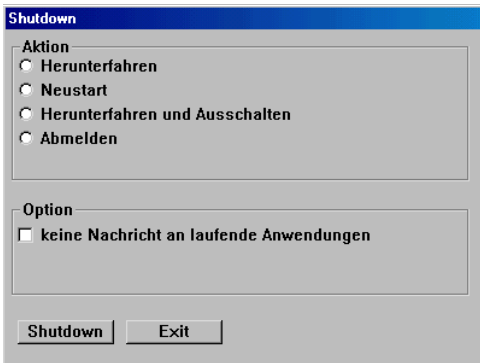


Bild 6.1: Die Anwendung nach dem Start

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT,  
                          WPARAM, LPARAM);
```

```
HINSTANCE hInstGlobal;  
HWND hButtonRadioButton, hButtonRadioButton2,  
     hButtonRadioButton3, hButtonRadioButton4,  
     hButtonCheckBox;
```

```
int APIENTRY WinMain(HINSTANCE hInstance,  
                    HINSTANCE hPrevInstance,  
                    LPSTR lpCmdLine,  
                    int nCmdShow )
```

```
{  
    hInstGlobal = hInstance;  
  
    WNDCLASS WndClass;  
    WndClass.style = 0;  
    WndClass.cbClsExtra = 0;  
    WndClass.cbWndExtra = 0;  
    WndClass.lpfnWndProc = WndProc;  
    WndClass.hInstance = hInstance;  
    WndClass.hbrBackground = (HBRUSH)(COLOR_MENU+1);
```

```

WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

int x,y;
x = GetSystemMetrics (SM_CXSCREEN);
y = GetSystemMetrics (SM_CYSCREEN);

HWND hWindow;
hWindow = CreateWindow("WinProg","Shutdown",
                      WS_OVERLAPPED,
                      (x/2)-200,(y/2),
                      150,400,300,NULL,NULL,
                      hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    TranslateMessage (&Message);
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_CREATE:

```

```
HWND hButtonShutdown, hButtonExit,
      hButtonGroupBox, hButtonGroupBox2;
hButtonShutdown =
    CreateWindow("BUTTON", "Shutdown",
                WS_CHILD | WS_VISIBLE |
                BS_PUSHBUTTON,
                10, 240, 80, 20,
                hWnd, (HMENU) 1,
                hInstGlobal, NULL);
hButtonExit =
    CreateWindow("BUTTON", "Exit",
                WS_CHILD |
                WS_VISIBLE |
                BS_PUSHBUTTON,
                100, 240, 80, 20,
                hWnd, (HMENU) 2,
                hInstGlobal, NULL);
hButtonGroupBox =
    CreateWindow("BUTTON", "Aktion",
                WS_CHILD |
                WS_VISIBLE |
                BS_GROUPBOX,
                5, 5, 380, 120,
                hWnd, (HMENU) 3,
                hInstGlobal, NULL);
hButtonRadioButton =
    CreateWindow("BUTTON", "Herunterfahren",
                WS_CHILD |
                WS_VISIBLE |
                BS_RADIOBUTTON,
                10, 20, 280, 20,
                hWnd, (HMENU) 4,
                hInstGlobal, NULL);
hButtonRadioButton2 =
    CreateWindow("BUTTON", "Neustart",
                WS_CHILD |
                WS_VISIBLE |
                BS_RADIOBUTTON,
```



```

        10,40,280,20,
        hWnd,(HMENU) 5,
        hInstGlobal, NULL);
hButtonRadioButton3 =
    CreateWindow("BUTTON",
        "Herunterfahren und Ausschalten",
        WS_CHILD |
        WS_VISIBLE |
        BS_RADIOBUTTON,
        10,60,280,20,
        hWnd,(HMENU) 6,
        hInstGlobal, NULL);
hButtonRadioButton4 =
    CreateWindow("BUTTON","Abmelden",
        WS_CHILD |
        WS_VISIBLE |
        BS_RADIOBUTTON,
        10,80,280,20,
        hWnd,(HMENU) 7,
        hInstGlobal, NULL);
hButtonGroupBox2 =
    CreateWindow("BUTTON","Option",
        WS_CHILD |
        WS_VISIBLE |
        BS_GROUPBOX,
        5,140,380,80,
        hWnd,(HMENU) 8,
        hInstGlobal, NULL);
hButtonCheckBox =
    CreateWindow("BUTTON",
        "keine Nachricht an laufende Anwendungen",
        WS_CHILD |
        WS_VISIBLE |
        BS_CHECKBOX,
        10,160,340,20,
        hWnd,(HMENU) 9,
        hInstGlobal, NULL);

return 0;

```

```
case WM_COMMAND:
    if (HIWORD(wParam) == BN_CLICKED)
    {
        if (LOWORD(wParam) == 1)
        {
            int Parameter;
            Parameter = 0;
            if (SendMessage (hButtonRadioButton,
                            BM_GETCHECK, 0, 0)
                == BST_CHECKED)
            {
                Parameter = EWX_SHUTDOWN;
            }
            if (SendMessage (hButtonRadioButton2,
                            BM_GETCHECK, 0, 0)
                == BST_CHECKED)
            {
                Parameter = EWX_REBOOT;
            }
            if (SendMessage (hButtonRadioButton3,
                            BM_GETCHECK, 0, 0) ==
                BST_CHECKED)
            {
                Parameter = EWX_POWEROFF;
            }
            if (SendMessage (hButtonRadioButton4,
                            BM_GETCHECK, 0, 0) ==
                BST_CHECKED)
            {
                Parameter = EWX_LOGOFF;
            }
            if (SendMessage (hButtonCheckBox,
                            BM_GETCHECK, 0, 0)
                == BST_CHECKED)
            {
                Parameter = Parameter | EWX_FORCE;
            }
            ExitWindowsEx (Parameter, 0);
        }
    }
}
```

```

}
if (LOWORD(wParam) == 2)
{
    SendMessage (GetParent((HWND)lParam),
                WM_DESTROY ,0 ,0);
}
if (LOWORD(wParam) == 4)
{
    if (!(SendMessage ((HWND)lParam,
                      BM_GETCHECK, 0, 0)
        == BST_CHECKED))
    {
        SendMessage ((HWND)lParam,
                    BM_SETCHECK, BST_CHECKED, 0);
        SendMessage (hButtonRadioButton2,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton3,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton4,
                    BM_SETCHECK, BST_UNCHECKED, 0);
    }
}
if (LOWORD(wParam) == 6)
{
    if (!(SendMessage ((HWND)lParam,
                      BM_GETCHECK, 0, 0)
        == BST_CHECKED))
    {
        SendMessage ((HWND)lParam,
                    BM_SETCHECK, BST_CHECKED, 0);
        SendMessage (hButtonRadioButton,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton2,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton4,
                    BM_SETCHECK, BST_UNCHECKED, 0);
    }
}

```

```
}
if (LOWORD(wParam) == 7)
{
    if (!(SendMessage ((HWND)lParam,
                        BM_GETCHECK, 0, 0)
        == BST_CHECKED))
    {
        SendMessage ((HWND)lParam,
                    BM_SETCHECK, BST_CHECKED, 0);
        SendMessage (hButtonRadioButton,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton3,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton2,
                    BM_SETCHECK, BST_UNCHECKED, 0);
    }
}
if (LOWORD(wParam) == 5)
{
    if (!(SendMessage ((HWND)lParam,
                        BM_GETCHECK, 0, 0)
        == BST_CHECKED))
    {
        SendMessage ((HWND)lParam,
                    BM_SETCHECK, BST_CHECKED, 0);
        SendMessage (hButtonRadioButton,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton3,
                    BM_SETCHECK, BST_UNCHECKED, 0);
        SendMessage (hButtonRadioButton4,
                    BM_SETCHECK, BST_UNCHECKED, 0);
    }
}
if (LOWORD(wParam) == 9)
{
    if (SendMessage ((HWND)lParam,
                    BM_GETCHECK, 0, 0)
        == BST_CHECKED)
```

```

        {
            SendMessage ((HWND)lParam,
                        BM_SETCHECK, BST_UNCHECKED, 0);
        }
        else
        {
            SendMessage ((HWND)lParam,
                        BM_SETCHECK, BST_CHECKED, 0);
        }
    }
}
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                        wParam, lParam);
}
}

```

## 6.2.2 Erläuterung des Quelltexts

### Der erste Teil der Anwendung

Im ersten Teil des Quelltexts wird wie gewohnt ein Fenster erstellt. Dieses Fenster wird aber nicht wie sonst am oberen linken Rand erstellt, sondern in der Mitte des Bildschirms positioniert. Dazu werden die Angaben über die Bildschirmgröße benötigt. Diese Angaben werden mit `GetSystemMetrics` ermittelt. Daraus errechnet sich dann die Position des Fensters, die es haben muss, um in der Mitte des Bildschirms zu erscheinen. Das Fenster bekommt diesmal die Eigenschaft `WS_OVERLAPPED`. Es ist ein Fenster, das von anderen überlappt werden kann und außerdem eine Titelleiste und einen Rand hat.

### Die Änderungen in der Fensterprozedur

Zunächst fällt auf, dass eine große Anzahl von Steuerelementen vom Typ `BUTTON` erstellt wird. Dabei werden drei neue Arten des Typs

BUTTON, nämlich die Styles `BS_RADIOBUTTON`, `BS_CHECKBOX` und `BS_GROUPBOX` verwendet. Die ersten beiden stellen markierbare Flächen dar. Die markierbaren Flächen werden meistens in `GroupBoxes`, also Gruppen, zusammengefasst. Von den `RadioButtons` kann nur jeweils einer aktiviert sein. Bei den `CheckBoxes` können auch mehrere gleichzeitig aktiviert sein. Ein Button vom Typ `BS_GROUPBOX` ist einfach nur ein Rahmen mit einer Überschrift für diese Buttons.

Weitere Änderungen erfolgen in der Nachricht `WM_COMMAND`. So muss das Umschalten der einzelnen `RadioButtons` gewährleistet werden. Dazu müssen die anderen `RadioButtons` alle unmarkiert sein, wenn einer markiert wird. Das Markieren muss man selber ausführen. Dieses gilt auch für die `CheckBoxes`.

Der eigentliche Hauptteil liegt aber in der Abfrage des Buttons `Shutdown`. Hier kommt die Funktion `ExitWindowsEx` zum Einsatz. Die Parameter der Funktion werden vorher kombiniert, so dass die Funktion nicht mehrmals aufgerufen werden muss. Diese Funktion wollen wir nun im Einzelnen betrachten.

```
BOOL ExitWindowsEx( UINT uFlags,  
                   DWORD dwReserved);
```

Sie taucht im Quelltext so auf:

```
ExitWindowsEx (Parameter, 0);
```

Die Funktion führt also verschiedene Aktionen aus, um den Rechner oder zumindest alle laufenden Anwendungen zu beenden, und um das Betriebssystem wieder neu zu initialisieren.

- `uFlags` bestimmt die Aktionen, die die Funktion `ExitWindowsEx` ausführt. Dabei treten fünf verschiedene Parameter auf. Die ersten vier Parameter können nicht miteinander kombiniert werden, aber der fünfte kann mit den vier anderen kombiniert werden.

`EWX_LOGOFF` meldet den Benutzer ab.

`EWX_POWEROFF` fährt den Rechner herunter und schaltet ihn bei entsprechender Hardwareunterstützung ab.

`EWX_REBOOT` startet den Rechner neu.

EWX\_SHUTDOWN fährt den Rechner herunter.

EWX\_FORCE bedeutet, dass keine Nachrichten an laufende Anwendungen gesendet werden. Die laufenden Anwendungen haben dann keine Möglichkeit ihre Daten zu sichern. Sie können so aber auch nicht den Prozess der Systemdeinitialisierung beeinflussen.

- `dwReserved` wird ignoriert.





# 7 Bitmaps

---

## 7.1 Allgemeines

Es gibt zwei Arten von Bitmaps, nämlich die DDBs (*device dependent bitmaps*, geräte-abhängige Bitmaps) und die DIBs (*device independent bitmaps*, geräte-unabhängige Bitmaps). Ein Bitmap ist eine Reihe von Daten, die Informationen über die Farbe von Pixeln eines Bilds enthalten. Die DDBs besitzen nur die Farbinformation der Pixel. Die DIBs verfügen über die Farbinformationen und zusätzlich über Angaben, wie diese zu interpretieren sind. In diesen Angaben finden sich u.a. Daten über Breite und Höhe des Bildes und über die Farbtiefe.

Sehr wichtig ist auch die Tatsache, dass Zeilen eines Bitmaps nur mit gradlinigen Byteangaben beendet werden dürfen, da sonst bei der Anzeige Fehler auftreten.

### 7.1.1 DDB

DDBs enthalten also nur die einfachen Pixeldaten. Sie sind Bitmaps, die zur Laufzeit der Anwendung erzeugt werden. Dies ist deshalb so, weil sie ja nur auf diesem Ausgabegerät angewendet werden sollen. Da diese Bitmaps keine Informationen über die Farbtiefe enthalten, können sie z. B. nicht auf einem Bildschirm mit anderer Farbtiefe abgebildet werden. Da die Funktion nicht weiß, wie das Bild in den Videospeicher soll, kopiert sie die Daten einfach. Dabei entsteht ein bunter Mischmasch.

DDBs werden wie GDI-Objekte behandelt. Es gibt hier ein Bitmap-Objekt. Dieses kann auch einem Gerätekontext-Objekt zugewiesen werden. Dieser Gerätekontext muss aber der Speicher sein. Das heißt es darf kein Gerätekontext des Bildschirms sein, denn dort läuft die Funktion nicht.

---

### 7.1.2 DIB

DIBs enthalten die Pixeldaten und weitere Angaben über die Pixel. Daher können DIBs auf jedem Bildschirm abgebildet werden. Wenn ein DIB eine höhere Farbtiefe hat als das Ausgabegerät, werden die Pixel anhand der Informationen über die Pixel dem Ausgabegerät angepasst.

### 7.1.3 Farbtiefe

Die Farbtiefe ist ein Wert, der angibt, wie viele Farben dargestellt werden können. Diese Einstellung kann man in der Systemsteuerung für den Bildschirm und für die Grafikkarte ändern. Die Farbtiefe stellt man ein, indem man angibt, wie viele Bits für einen Pixel bereitstehen sollen. Fast alle Monitore – bis auf sehr alte – können RGB-Farben darstellen. Doch die Grafikkarten haben damit manchmal Probleme, weil sie nicht über genügend Speicher verfügen. Die Anzahl der darstellbaren Farben errechnet sich über  $2^x$  Bits. Falls man also  $2^8$  Bits hat, sind dies 256 verschiedene Farbwerte. Bei dieser Zahl werden allerdings die RGB-Werte aus einer Farbtabelle entnommen, in die der Wert der Pixel nur einen Index darstellt. Bei  $2^{24}$  Bits hat man 16,7 Millionen verschiedene Farbwerte. Dieser Wert entspricht TrueColor. Es gibt hier keine Farbtabelle, sondern nur 24 Bits pro Pixel, wobei jeweils 8 Bit den Wert für Blau, Rot und Grün angeben. Wie die Pixeldaten zu interpretieren sind, muss der Grafikkarte vorher mitgeteilt werden. Dies geschieht über die Systemsteuerung. Die Grafikkarte wandelt dann die digitalen Signale in analoge um, die der Monitor anzeigen kann. Bei 256 Farbwerten muss der Grafikkarte zuerst also eine Farbtabelle übermittelt werden. Andere Farbtiefen als 4, 8, 16 und 32 können nicht eingestellt werden, denn die GDI-Funktionen von Windows müssen mit diesen Werten zurechtkommen. Die GDI-Funktionen passen Bilder mit mehr Farben einer geringeren Farbtiefe an. Alle Farbwerte in GDI-Funktionen werden grundsätzlich über 32-Bit-Werte angegeben. Dabei bleiben die ersten 8 Bit ungenutzt. Dann folgen jeweils 8 Bit mit den RGB-Werten. Diese RGB-Werte werden von den GDI-Funktionen der Farbtiefe des Ausgabegerätes angepasst. Festzuhalten bleibt, dass bei den meisten Grafikkarten auf gleiche Weise definiert ist, wie eine Anzahl von Bits pro Pixel interpretiert werden soll, um einen analogen Farbwert an den Monitor zu senden. Die GDI-Funktionen reagieren nur auf die Standards.

## 7.2 Die DDB-Anwendung

### 7.2.1 Der Quelltext

In der folgenden Anwendung wird ein DDB erzeugt. DDBs werden wie GDI-Objekte behandelt. Als Erstes wird ein Speicher-Gerätekontext-Objekt erzeugt. Dies geschieht durch die Funktion `CreateCompatibleDC`. Dann wird das Bitmap-Objekt erzeugt. Dieses enthält dann die Pixeldaten des DDBs und wird dem Speicher-Gerätekontext-Objekt zugewiesen. Anschließend werden die Daten aus dem Speicher-Gerätekontext-Objekt dem Bildschirm-Gerätekontext-Objekt mit der Funktion `BitBlt` zugewiesen. Damit das Programm funktioniert, muss die Farbtiefe auf 8 Bit eingestellt werden. Benutzen Sie dazu die Systemsteuerung. Klicken Sie zweimal auf das Anzeigesymbol und danach auf den Reiter Einstellungen und ändern Sie den Farbwert in 256.



Bild 7.1: Anwendung nach dem Start



Bild 7.2: Anwendung nach einem Klick mit der linken Maustaste

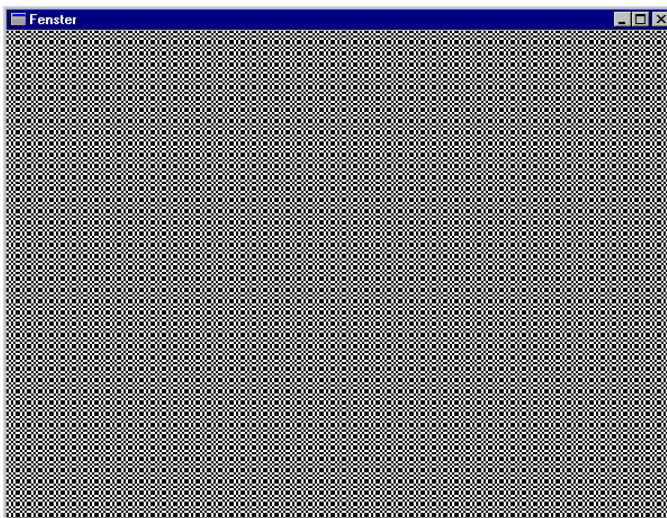


Bild 7.3: nach einem Klick mit der rechten Maustaste

```

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWnd;
    hWnd = CreateWindow("WinProg","Fenster",
                      WS_OVERLAPPEDWINDOW,
                      0,0,600,460,NULL,NULL,
                      hInstance, NULL);

    ShowWindow (hWnd, nCmdShow);

    UpdateWindow (hWnd);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
    {
        DispatchMessage(&Message);
    }
}

```

```
        return (Message.wParam);
    }

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    static BYTE Pixel[6][5] = {0 ,255,0 ,255,0 ,0,
                              255,0 ,255,0 ,255,0,
                              0 ,255,0 ,255,0 ,0,
                              255,0 ,255,0 ,255,0,
                              0 ,255,0 ,255,0 ,0};

    static HBITMAP hBitmap;
    static HDC hdcmem;

    switch(uiMessage)
    {
        case WM_LBUTTONDOWN:
            HDC hdc;
            hdc = GetDC (hWnd);

            hdcmem = CreateCompatibleDC (hdc);
            hBitmap = CreateBitmap (6,5,1,8,Pixel);
            SelectObject (hdcmem, hBitmap);

            RECT rect;
            GetClientRect (hWnd, &rect);
            int x,y;
            for (y=0;y<rect.bottom;y=y+5)
            {
                for (x=0;x<rect.right;x=x+5)
                {
                    BitBlt (hdc,x,y,5,5,hdcmem,0,0,SRCCOPY);
                }
            }

            DeleteObject (hBitmap);
            DeleteDC (hdcmem);
    }
}
```

```

    ReleaseDC (hWnd, hdc);
    return 0;
case WM_RBUTTONDOWN:
    hdc = GetDC (hWnd);

    hdcmem = CreateCompatibleDC (hdc);
    hBitmap = CreateBitmap (5,5,1,8,Pixel);
    SelectObject (hdcmem, hBitmap);

    GetClientRect (hWnd, &rect);
    for (y=0;y<rect.bottom;y=y+10)
    {
        for (x=0;x<rect.right;x=x+10)
        {
            StretchBlt (hdc,x,y,10,10,
                        hdcmem,0,0,5,5,SRCCOPY);
        }
    }

    DeleteObject (hBitmap);
    DeleteDC (hdcmem);
    ReleaseDC (hWnd, hdc);
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}

```

## 7.2.2 Besprechung des Quelltexts

Veränderungen haben sich nur in der Funktion `WndProc` ergeben.

### Neue Variablen

Es werden mehrere neue Variablen deklariert.

```
static BYTE Pixel[6][5] = {0 ,255,0 ,255,0 ,0,
                          255,0 ,255,0 ,255,0,
                          0 ,255,0 ,255,0 ,0,
                          255,0 ,255,0 ,255,0,
                          0 ,255,0 ,255,0 ,0};
```

```
static HBITMAP hBitmap;
static HDC hdcmem;
```

- Der Pixel-Array enthält die Daten der Pixel. Das Bild hat eigentlich 5x5 Pixel. Eine Pixelzeile muss mit einer gradlinigen Anzahl von Pixeln abgeschlossen werden. Das Bitmap hat deshalb 6x5 Pixel. Dies sind die eigentlichen Pixeldaten.
- `hBitmap` ist der Handle auf das Bitmap-Objekt. Dieses wird später erstellt.
- `hdcmem` ist ein ganz normaler Handle auf ein Gerätekontext-Objekt. Aber diesmal ist der Gerätekontext kein Teil des Bildschirms, sondern nur ein Teil des Speichers, der einen grafischen Gerätekontext simuliert.

7

## Die Nachricht WM\_LBUTTONDOWN

Die Nachricht WM\_LBUTTONDOWN erhält ein Fenster, wenn in ihr die linke Maustaste gedrückt wird. Anschließend wird die Hauptfunktion des Programms ausgeführt.

```
HDC hdc;
hdc = GetDC (hWnd);

hdcmem = CreateCompatibleDC (hdc);
hBitmap = CreateBitmap (6,5,1,8,Pixel);
SelectObject (hdcmem, hBitmap);

RECT rect;
GetClientRect (hWnd, &rect);
int x,y;
for (y=0;y<rect.bottom;y=y+5)
{
    for (x=0;x<rect.right;x=x+5)
    {
        BitBlt (hdc,x,y,5,5,hdcmem,0,0,SRCCOPY);
```



```
}  
}
```

```
DeleteDC (hdcmem);  
ReleaseDC (hWnd, hdc);  
return 0;
```

### *CreateCompatibleDC*

Als Erstes wird der Gerätekontext-Handle des Hauptfensters ermittelt. Danach wird ein Speicher-Gerätekontext-Objekt erzeugt. Dieser Speicher-Gerätekontext soll mit dem Gerätekontext des Fensters kompatibel sein. All dies erreichen wir mit der Funktion `CreateCompatibleDC`.

```
HDC CreateCompatibleDC( HDC hdc);
```

Im Quelltext wird dieser Befehl zweimal auf die gleiche Weise verwendet.

```
hdcmem = CreateCompatibleDC (hdc);
```

- `hdc` ist der Handle auf das Gerätekontext-Objekt, von dem das Speicher-Gerätekontext-Objekt erzeugt werden soll.

### *CreateBitmap*

Als Nächstes wird mit der Funktion `CreateBitmap` ein Bitmap-Objekt erzeugt. Das Bitmap-Objekt beinhaltet dann die Daten des Bitmaps, also die Pixel. Da auf diese Weise ein DDB erzeugt wird, enthält dieses Objekt keine Daten darüber, wie, also mit welcher Farbtiefe, die Pixel abgespeichert wurden. Diese Werte müssen aber angegeben werden.

```
HBITMAP CreateBitmap( int nWidth, int nHeight,  
                     UINT cPlanes, UINT cBitsPerPel,  
                     CONST VOID *lpvBits);
```

Im Quelltext ist diese Funktion zweimal vorhanden.

```
hBitmap = CreateBitmap (6,5,1,8,Pixel);
```

- `nWidth` und `nHeight` geben die Breite und Höhe des Bitmaps (DDB) an.

- `cPlanes` ist die Anzahl der Farbebenen. Dieser Wert ist uninteressant und wird deswegen auf 1 gesetzt.
- `cBitsPerPixel` ist ein Wert, der die Anzahl der Bits pro Pixel angibt.
- `lpvBits` ist ein Zeiger auf den Speicherbereich, in dem die Pixel abgespeichert sind.

### *GetClientRect*

Anschließend wird das Bitmap-Objekt dem Gerätekontext-Objekt zugewiesen. Danach wird eine Struktur vom Typ `RECT` deklariert. Diese Struktur wird für die Funktion `GetClientRect` benötigt. Mit ihr werden die Daten über das Rechteck des Gerätekontext-Objekts des Fensters abgerufen. Diese Daten werden in Pixeln angegeben und beziehen sich auf die obere linke Ecke des Gerätekontexts.

```
BOOL GetClientRect( HWND hWnd, LPRECT lpRect);
```

Im Quelltext ist diese Funktion ebenfalls zweimal vorhanden.

```
GetClientRect( hWnd, &rect);
```

- `hWnd` ist der Handle auf das Fenster-Objekt, von dem die Maße des Gerätekontext-Objektes ermittelt werden sollen.
- `lpRect` ist ein Zeiger auf eine Struktur vom Typ `RECT`, in dem die Maße des Gerätekontext-Objektes abgespeichert werden.

### *BitBlt*

Als Nächstes werden Daten des Speicher-Gerätekontext-Objekts in das Gerätekontext-Objekt des Bildschirms kopiert. Dies geschieht mit der Funktion `BitBlt`. Da dieses Bitmap die ganze Fläche des Gerätekontext-Objekts ausfüllen soll, wird es mehrmals in regelmäßigen Abständen in das Gerätekontext-Objekt des Bildschirms kopiert.

```
BOOL BitBlt( HDC hdcDest, int nXDest, int nYDest,
            int nWidth, int nHeight, HDC hdcSrc,
            int nXSrc, int nYSrc, DWORD dwRop);
```

Diese Funktion kommt zweimal im Quelltext vor.

```
BitBlt( hdc, x, y, 5, 5, hdcmem, 0, 0, SRCCOPY);
```

- `hdcDest` ist der Handle auf das Gerätekontext-Objekt, in das die Bilddaten kopiert werden sollen.

- `nXDest` und `nYDest` geben die Position an, wohin die Bilddaten kopiert werden sollen.
- `nWidth` und `nHeight` sind Breite und Höhe der Bilddaten, die kopiert werden sollen. Sie beziehen sich sowohl auf das Quell- als auch auf das Ziel-Objekt.
- `hdcSrc` ist der Handle auf das Gerätekontext-Objekt, aus dem die Daten kopiert werden.
- `nXSrc` und `nYSrc` ist die Position im Quell-Objekt, von der aus die Bilddaten kopiert werden sollen.
- `dwRop` ist ein Wert, der mit Konstanten belegt wird. Eine davon ist unter dem Namen `SRCCPY` deklariert. Diese Konstante besagt, dass das Quell-Rechteck mit Bilddaten auf das Ziel-Rechteck kopiert wird.

### *DeleteDC*

Jetzt löschen wir noch das Speicher-Gerätekontext-Objekt. Dazu verwenden wir die Funktion `DeleteDC`. Danach wird der Handle auf das Gerätekontext-Objekt des Bildschirms wieder freigegeben.

```
BOOL DeleteDC( HDC hdc);
```

Im Quelltext hat die Funktion folgendes Aussehen:

```
DeleteDC (hdcmem);
```

- `hdc` ist der Handle auf das Gerätekontext-Objekt, das gelöscht werden soll.

### **Die Nachricht WM\_RBUTTONDOWN**

Wenn diese Nachricht erscheint, passiert genau das Geiche wie bei `WM_LBUTTONDOWN`. Einen Unterschied gibt es allerdings: Das Kopieren der Pixeldaten erfolgt durch die Funktion `StretchBlt`. Dadurch werden die Pixeldaten vergrößert dargestellt.

```
BOOL StretchBlt(HDC hdcDest, int nXOriginDest,
               int nYOriginDest, int nWidthDest,
               int nHeightDest, HDC hdcSrc,
               int nXOriginSrc, int nYOriginSrc,
               int nWidthSrc, int nHeightSrc,
               DWORD dwRop);
```

Diese Funktion tritt im Quelltext wie folgt auf.

```
StretchBlt (hdc,x,y,10,10,  
           hdcmem,0,0,5,5,SRCCOPY);
```

- `hdcDest` ist der Handle auf das Gerätekontext-Objekt, in das die Bilddaten kopiert werden sollen.
- `nXOriginDest` und `nYOriginDest` geben die Position an, wohin die Bilddaten kopiert werden sollen.
- `nWidthDest` und `nHeightDest` sind Breite und Höhe der Bilddaten, die kopiert werden sollen. Diese beziehen sich auf das Ziel-Objekt. Sie geben damit auch gleich die Vergrößerung der Bilddaten an.
- `hdcSrc` ist der Handle auf das Gerätekontext-Objekt aus dem die Daten kopiert werden.
- `nXOriginSrc` und `nYOriginSrc` ist die Position im Quell-Objekt, von der aus die Bilddaten kopiert werden sollen.
- `nWidthSrc` und `nHeightSrc` sind Breite und Höhe der Bilddaten, die kopiert werden sollen. Diese beziehen sich auf das Quell-Objekt.
- `dwRop` ist ein Wert, der mit Konstanten belegt wird. Eine davon ist unter dem Namen `SRCCPY` deklariert. Diese Konstante besagt, dass das Quell-Rechteck mit Bilddaten auf das Ziel-Rechteck kopiert wird.

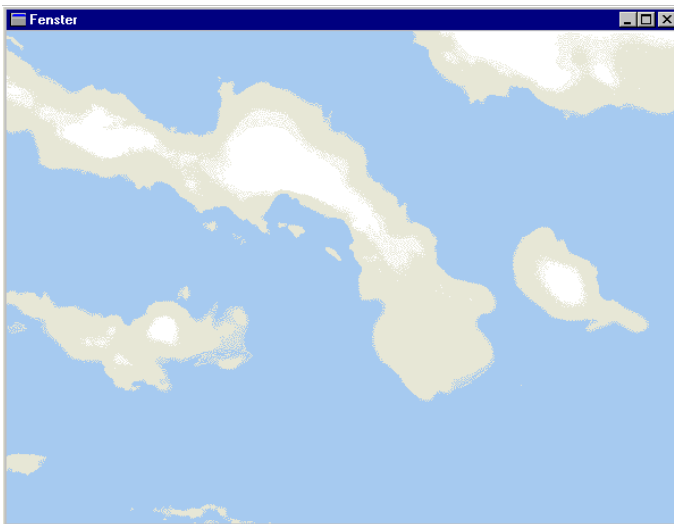
## 7.3 Die DIB-Anwendung

### 7.3.1 Der Quelltext

Diese Anwendung lädt eine Bitmap-Datei, also eine BMP-Datei, von der Festplatte in den Speicher. Es ist also eine DIB-Datei, wie sie von Paintbrush auch erzeugt wird. Da ein DIB aus mehreren Strukturen besteht, wird den einzelnen Strukturen Speicherplatz aus dem Arbeitsspeicher zugewiesen. Die Bilddaten werden dem Ausgabegerät angepasst und dann auf den Bildschirm übertragen. Danach wird der Speicher wieder freigesetzt.



*Bild 7.4: Anwendung nach dem Start*



*Bild 7.5: Anwendung nach einem Klick auf die linke Maustaste*

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
    hWindow = CreateWindow("WinProg", "Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0, 0, 600, 460, NULL, NULL,
                          hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
    {
        DispatchMessage(&Message);
    }
}
```

```

    return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    static BITMAPFILEHEADER *pbmfh;
    static BITMAPINFO *pbmi;
    static BYTE *pBits;
    static int cxDib, cyDib;

    switch(uiMessage)
    {
        case WM_LBUTTONDOWN:
            DWORD dwFileSize, dwHighSize, dwBytesRead;
            HANDLE hFile;
            hFile = CreateFile ("C:\\test.bmp", GENERIC_READ,
                               FILE_SHARE_READ, NULL,
                               OPEN_EXISTING,
                               FILE_FLAG_SEQUENTIAL_SCAN,
                               NULL);
            dwFileSize = GetFileSize (hFile, &dwHighSize);
            pbmfh = (BITMAPFILEHEADER *) malloc (dwFileSize);
            ReadFile (hFile, pbmfh, dwFileSize,
                    &dwBytesRead, NULL);
            pbmi = (BITMAPINFO *) (pbmfh + 1);
            pBits = (BYTE *) pbmfh + pbmfh->bfOffBits;
            cxDib = pbmi->bmiHeader.biWidth;
            cyDib = abs(pbmi->bmiHeader.biHeight);
            HDC hdc;
            hdc = GetDC (hWnd);

            SetDIBitsToDevice (hdc,
                              0,
                              0,
                              cxDib,
                              cyDib,

```

```

        0,
        0,
        0,
        cyDib,
        pBits,
        pbmi,
        DIB_RGB_COLORS);

    ReleaseDC (hWnd, hdc);
    free (pbmfh);
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}

```

7

### 7.3.2 Besprechung des Quelltexts

Die ersten Änderungen in diesem Quelltext finden sich in der Nachricht WM\_LBUTTONDOWN.

#### Neue Variablen

```

static BITMAPFILEHEADER *pbmfh;
static BITMAPINFO *pbmi;
static BYTE *pBits;
static int cxDib, cyDib;

```

Es werden zwei neue Zeiger auf Strukturen angelegt. Die Strukturen sind vom Typ BITMAPFILEHEADER und BITMAPINFO. Es wird ein Zeiger auf Variablen des Typs Byte angelegt. Danach werden noch zwei richtige Variablen deklariert.

Die BITMAPFILEHEADER Struktur dient dazu, Informationen über Typ, Größe und Layout eines DIB zu speichern.



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

- `bfType` spezifiziert den Dateityp. Wir sprechen von einem Dateityp, da DIBs gewöhnlich aus Dateien geladen werden. Der Dateityp muss „BM“ sein.
- `bfSize` ist die Größe des Bitmap-Files in Bytes.
- `bfReserved1` und `bfReserved2` sollen beide NULL sein.
- `bfOffBits` ist die Anzahl der Bytes vom Anfang der DIB-Datei bis zu den eigentlichen Pixeldaten gezählt.

Die `BITMAPINFO` Struktur enthält Informationen über die Größe und Farbtiefe des Bitmaps. Das Bitmap ist natürlich ein DIB.

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

- Beide Variablen der Struktur `BITMAPINFO` sind selbst Strukturen.

## Die Datei wird eingelesen

Als Nächstes muss die DIB Datei eingelesen werden. Die Funktion zur Arbeit mit Dateien werden hier jetzt nicht genau beschrieben. Festzuhalten bleibt, dass die Datei in einen vorher reservierten Speicher eingelesen wird. Der Anfang dieses Speichers bildet dann den Anfang der Struktur `pbfh`.

```
DWORD dwFileSize, dwHighSize, dwBytesRead;
HANDLE hFile;
hFile = CreateFile ("C:\\test.bmp", GENERIC_READ,
                  FILE_SHARE_READ, NULL,
                  OPEN_EXISTING,
```

```
FILE_FLAG_SEQUENTIAL_SCAN,  
NULL);
```

```
dwFileSize = GetFileSize (hFile, &dwHighSize);  
pbmfh = (BITMAPFILEHEADER *) malloc (dwFileSize);  
ReadFile (hFile, pbmfh, dwFileSize,  
          &dwBytesRead, NULL);
```

## Die Strukturen werden festgelegt

Der Zeiger von `pbmfh` ist sowieso auf den Anfang der eingelesenen Datei gesetzt. Damit wäre diese Strukturen schon einmal belegt. Die nächste Struktur folgt direkt anschließend. Sie ist vom Typ `BITMAPINFO`. Diese Struktur kann auch eine andere sein, das hängt von der jeweiligen Windowsversion ab. (Es kann auch eine ganz andere sein, wie z.B. in der OS/2 Urversion.) Jetzt wird noch die Position der Pixeldaten festgelegt. Dann werden bestimmte Daten aus diesen Strukturen in Variablen abgespeichert, dazu gehören die Höhe und Breite des Bilds.

```
pbmi = (BITMAPINFO *) (pbmfh + 1);  
pBits = (BYTE *) pbmfh + pbmfh->bfoffBits;  
cxDib = pbmi->bmiHeader.biWidth;  
cyDib = abs(pbmi->bmiHeader.biHeight);
```

## SetDIBitsToDevice

Nun wird ein Handle auf das Gerätekontext-Objekt ermittelt, in dem das Bitmap angezeigt werden soll. Dann wird mit der Funktion `SetDIBitToDevice` eine Anpassung der Pixel an die Farbtiefe des Ausgabegerätes vorgenommen. Die Pixel werden dann auf dem Ausgabegerät abgebildet.

```
int SetDIBitsToDevice( HDC hdc, int XDest, int YDest,  
                      DWORD dwWidth, DWORD dwHeight,  
                      int XSrc, int YSrc,  
                      UINT uStartScan, UINT cScanLines,  
                      CONST VOID *lpvBits,  
                      CONST BITMAPINFO *lpbmi,  
                      UINT fuColorUse);
```

Im Quelltext sieht diese Funktion wie folgt aus.

7

```
SetDIBitsToDevice (hdc,  
                  0,  
                  0,  
                  cxDib,  
                  cyDib,  
                  0,  
                  0,  
                  0,  
                  cyDib,  
                  pBits,  
                  pbmi,  
                  DIB_RGB_COLORS);
```

- `hdc` ist der Handle auf das Gerätekontext-Objekt.
- `Xdest` und `Ydest` geben die relative Position im Zielgerät in Pixeln an.
- `dwWidth` und `dwHeight` sind Breite und Höhe des Bitmaps.
- `XSrc` und `YSrc` geben die Position in der Quell-Struktur an.
- `uStartScan` ist die erste ScanLine im DIB.
- `cScanLine` ist die Anzahl der Scan-Lines, die im DIB vorhanden sind.
- `lpvBits` ist der Zeiger auf die Pixeldaten.
- `lpbmi` ist der Zeiger auf die Struktur BITMAPINFO.
- `fuColorUse` ist eine Konstante `DIB_RGB_COLORS`. Sie besagt, dass hier ganz normale RGB-Werte eingesetzt werden sollen.



## 8.1 Allgemeines

Dieses Kapitel beschäftigt sich mit dem Erstellen von Menüs für Fenster. Ein Menü ist eine Liste von Punkten, die die Aktionen einer Anwendung darstellen. Die Menüleiste befindet sich fast immer unter der Titelleiste. Man kann sie anklicken, wodurch bestimmte Aktionen für das Programm ausgelöst werden.

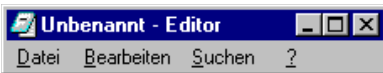


Bild 8.1: Menüleiste des Editors unter der Titelleiste

Ein Menü ist kein Fenster-, sondern ein Menüobjekt, auf das das Fensterobjekt zugreift. Ein Menüobjekt gehört zu einem Fensterobjekt. Ein Fensterobjekt zeichnet sich dadurch aus, dass es eine Funktion zur Nachrichtenverarbeitung besitzt, die von der Anwendung aufgerufen wird, um das Fenster anzuzeigen und zu verwalten. Ein Menüobjekt ist also einfach eine Datenstruktur, die sich nicht selber zeichnet, sondern von dem Fensterobjekt genutzt wird. Denn die Funktion des Fensterobjekts nutzt die Informationen des Menüobjekts, um ein Menü in das Fenster zu zeichnen.

Eine Menüleiste wird durch eine Reihe von Objekten erstellt. Man erstellt also Menüobjekte und weist ihnen Einträge zu. Dabei kann ein Menüeintrag eine Funktion auslösen oder auf ein Untermenü verweisen. So entsteht eine Hierarchie. Ein Menü wird erstellt und wird mit Menüpunkten gefüllt. Alle Menüpunkte, die eine Aktion auslösen sollen, bekommen dabei eine Identifikationsnummer. Dann werden neue Menüobjekte erstellt. Diesmal handelt es sich um Untermenüobjekte, die wiederum Menüpunkte enthalten und sich eigentlich nicht von den anderen Menüobjekten unterscheiden. Die Menüpunkte können ebenfalls eine Aktion auslösen. Diese Untermenüob-



jekte weist man als Menüeintrag dem vorher erstellten Menüobjekt zu. Das erste Objekt eines Menüs ist immer ein normales Menüobjekt. Alle anderen Menüobjekte sind Untermenüobjekte. Durch die Verbindung mit Menüpunkten entsteht eine Hierarchie. Der oberste Punkt der Hierarchie (also das Menüobjekt) wird dem Fenster zugewiesen.

Das oberste Menüobjekt wird in der Menüleiste angezeigt. Die Menüleiste des Fensters ist immer im Fenster sichtbar, die Untermenüs aber sind nicht sichtbar. Einträge in Menüleisten sollten keine Funktionen auslösen, sondern sie sollten auf Untermenüs verweisen. Natürlich ist es auch möglich in der Menüleiste Menüpunkte zu haben, die Funktionen auslösen.

Nur Fenster vom Typ `WS_OVERLAPPED` und `WS_POPUP` können Menüs enthalten.

Die Aktionen, die Menüeinträge auslösen können, werden durch die Nachrichtenverarbeitung des Fensters gesteuert. Wenn ein Menüpunkt ausgewählt wurde, wird eine Nachricht für das Fenster in die Warteschlange eingetragen, zu dem das Menü gehört. Sie enthält die Identifizierungsnummer des Menüeintrages.

Menüpunkte können durch Standardtasten ausgelöst werden. Dieser Service wird automatisch bereitgestellt.

Ein Menüobjekt muss nicht von der Anwendung gelöscht werden, bevor sie beendet wird. Solange das Menüobjekt mit dem Fensterobjekt verbunden ist, wird es gelöscht, wenn das Fensterobjekt gelöscht wird. Falls das Menüobjekt nicht mit einem Fensterobjekt verbunden ist, muss es am Ende der Anwendung gelöscht werden.

## 8.2 Die Anwendung

### 8.2.1 Der Quelltext

In diesem Programm wird ein Fenster erstellt und angezeigt. Dann wird eine Hierarchie von Menüobjekten erstellt. Das oberste Menüobjekt wird dem Fensterobjekt zugewiesen. Damit hat das Programm ein Menü. Der Aufbau des Menüs ist in der folgenden Tabelle zu sehen.

Menüpunkt 1. Ordnung	Menüpunkt 2. Ordnung	Menüpunkt 3. Ordnung	Aktion oder Verweis
Zeichne Rechteck			
Datei			
	Beenden		
Information			
	Autor		
	Grafik		
		Farbtiefe	
		Auflösung	
	Festplatte		
		Windows- Verzeichnis	
		Aktuelles Verzeichnis	

Tabelle 8.1: Das Menü der Anwendung

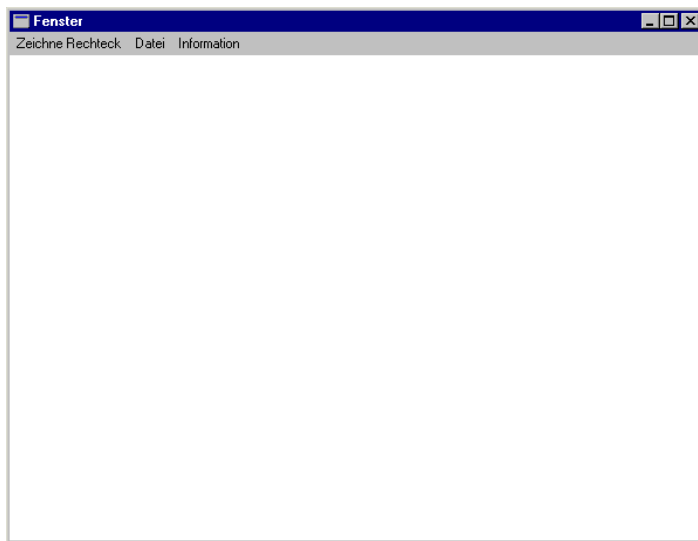


Bild 8.2: Die Anwendung nach dem Start



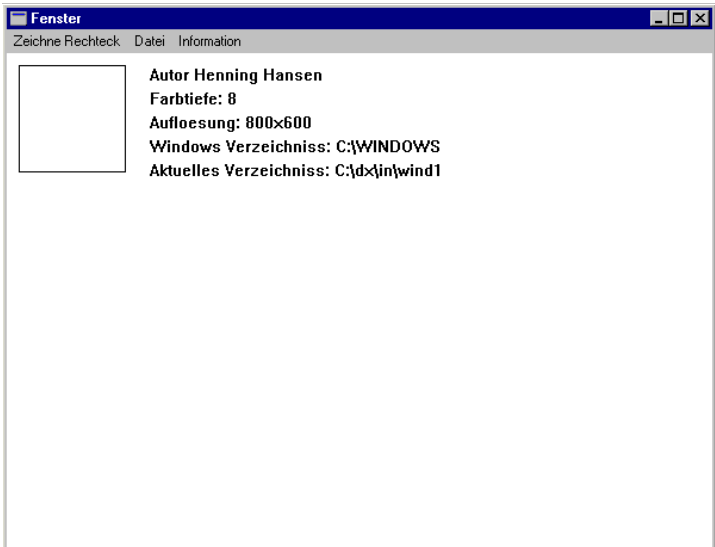


Bild 8.3: Die Anwendung nach der Ausführung der meisten Menüpunkte

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

```
int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
```

```
{
```

```
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfnWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
```



```

WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "Prog8";

RegisterClass(&WndClass);

HWND hWindow;
hWindow = CreateWindow("Prog8","Fenster",
                      WS_OVERLAPPEDWINDOW,
                      0,0,600,460,NULL,NULL,
                      hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,

                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_CREATE:
            HMENU hMenu;
            hMenu = CreateMenu ();
            MENUITEMINFO mii;
            mii.cbSize = sizeof(MENUITEMINFO);
            mii.fMask = MIIM_TYPE | MIIM_ID;
            mii.fType = MFT_STRING;

```



```

char *string;
string = new char[40];
lstrcpy (string,"Zeichne Rechteck");
mii.dwTypeData = string;
mii.cch = lstrlen (string);
mii.fState = MFS_ENABLED;
mii.wID = 1;
InsertMenuItem (hMenu, 0, FALSE, &mii);

HMENU hMenu2;
hMenu2 = CreatePopupMenu ();
lstrcpy (string, "Beenden");
mii.wID = 2;
InsertMenuItem (hMenu2, 0, FALSE, &mii);

lstrcpy (string, "Datei");
mii.fMask = MIIM_TYPE | MIIM_SUBMENU;
mii.hSubMenu = hMenu2;
InsertMenuItem (hMenu, 0, FALSE, &mii);

```

```

HMENU hMenu2_;
hMenu2_ = CreatePopupMenu ();
lstrcpy (string, "Autor");
mii.fMask = MIIM_TYPE | MIIM_ID;
mii.wID = 3;
InsertMenuItem (hMenu2_, 0, FALSE, &mii);

```

```

HMENU hMenu3;
hMenu3 = CreatePopupMenu ();
lstrcpy (string, "Farbtiefe");
mii.fMask = MIIM_TYPE | MIIM_ID;
mii.wID = 4;
InsertMenuItem (hMenu3, 0, FALSE, &mii);
lstrcpy (string, "Aufloesung");
mii.wID = 5;
InsertMenuItem (hMenu3, 0, FALSE, &mii);

```

```

HMENU hMenu3_;

```

```

hMenu3_ = CreatePopupMenu ();
lstrcpy (string, "Windows Verzeichniss");
mii.wID = 6;
InsertMenuItem (hMenu3_, 0, FALSE, &mii);
lstrcpy (string, "Aktuelles Verzeichniss");
mii.wID = 8;
InsertMenuItem (hMenu3_, 0, FALSE, &mii);

lstrcpy (string, "Grafik");
mii.fMask = MIIM_TYPE | MIIM_SUBMENU;
mii.hSubMenu = hMenu3;
InsertMenuItem (hMenu2_, 0, FALSE, &mii);
lstrcpy (string, "Festplatte");
mii.hSubMenu = hMenu3_;
InsertMenuItem (hMenu2_, 0, FALSE, &mii);

lstrcpy (string, "Information");
mii.hSubMenu = hMenu2_;
InsertMenuItem (hMenu, 0, FALSE, &mii);

SetMenu (hWnd, hMenu);

delete string[];

return 0;
case WM_COMMAND:
if (HIWORD(wParam) == 0)
{
switch (LOWORD(wParam))
{
case 1:
HDC hdc;
hdc = GetDC (hWnd);
Rectangle (hdc, 10, 10, 100, 100);
ReleaseDC(hWnd, hdc);
return 0;
case 2:

```



```
        DestroyWindow (hWnd);
        return 0;
case 3:
    hdc = GetDC (hWnd);
    char *string1;
    string1 = new char[80];
    lstrcpy (string1, "Autor Henning Hansen");
    TextOut (hdc, 120, 10, string1,
            lstrlen(string1));
    ReleaseDC(hWnd, hdc);
    delete string1[];
    return 0;
case 4:
    hdc = GetDC (hWnd);
    string1 = new char[80];
    lstrcpy (string1, "Farbtiefe: ");
    int BitsPixel;
    BitsPixel = GetDeviceCaps (hdc, BITSPIXEL);
    char *string2;
    string2 = new char[80];
    itoa (BitsPixel, string2,10);
    lstrcat (string1, string2);
    TextOut (hdc, 120, 30, string1,
            lstrlen(string1));
    ReleaseDC(hWnd, hdc);
    delete string1[];
    delete string2[];
    return 0;
case 5:
    hdc = GetDC (hWnd);
    string1 = new char[80];
    lstrcpy (string1, "Aufloesung: ");
    int HRes, VRes;
    HRes = GetDeviceCaps (hdc, HORZRES);
    string2 = new char[80];
    itoa (HRes, string2,10);
    lstrcat (string1, string2);
    lstrcat (string1, "x");
```

```

VRes = GetDeviceCaps (hdc, VERTRES)
itoa (VRes, string2,10);
lstrcat (string1, string2);
TextOut (hdc, 120, 50, string1,
lstrlen(string1));
ReleaseDC(hWnd, hdc);
delete string1[];
delete string2[];
return 0;
case 6:
hdc = GetDC (hWnd);
string1 = new char[80];
lstrcpy (string1, "Windows Verzeichniss: ");
string2 = new char[80];
GetWindowsDirectory (string2, 500);
lstrcat (string1, string2);
TextOut (hdc, 120, 70, string1,
lstrlen(string1));
ReleaseDC(hWnd, hdc);
delete string1[];
delete string2[];
return 0;
case 8:
hdc = GetDC (hWnd);
string1 = new char[80];
lstrcpy (string1,
"Aktuelles Verzeichniss:");
string2 = new char[80];
GetCurrentDirectory (500, string2);
lstrcat (string1, string2);
TextOut (hdc, 120, 90, string1,
lstrlen(string1));
ReleaseDC(hWnd, hdc);
delete string1[];
delete string2[];
return 0;
default:
return 0;

```



```

    }

    }
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}

```

### 8.2.2 Erläuterung

Im ersten Teil der Anwendung findet sich eigentlich nur Bekanntes. Die Änderungen und Neuerungen liegen wie fast immer in der Fensterfunktion. Die ersten und wichtigsten Änderungen gibt es bei der Nachrichtenbehandlung `WM_CREATE`. Dort wird das Menü erstellt und dem Fenster zugewiesen.

#### Die Erstellung des Menüs

In diesem Teil wird besprochen, wie das Menü erstellt wird. Dazu werden Menüobjekte angelegt. Diese werden mit Menüeinträgen gefüllt. Einige dieser Menüeinträge werden als Verweise auf andere Menüobjekte benutzt. Dann wird dem Windowsfenster ein Menüobjekt zugeordnet. Dieses Objekt ist normalerweise das oberste in der Hierarchie, damit alle anderen Menüobjekte von ihm aufgerufen werden können.

```

HMENU hMenu;
hMenu = CreateMenu ();
MENUITEMINFO mii;
mii.cbSize = sizeof(MENUITEMINFO);
mii.fMask = MIIM_TYPE | MIIM_ID;
mii.fType = MFT_STRING;
char *string;
string = new char[40];
lstrcpy (string,"Zeichne Rechteck");

```

```
mii.dwTypeData = string;
mii.cch = lstrlen (string);
mii.fState = MFS_ENABLED;
mii.wID = 1;
InsertMenuItem (hMenu, 0, FALSE, &mii);
```

```
HMENU hMenu2;
hMenu2 = CreatePopupMenu ();
lstrcpy (string, "Beenden");
mii.wID = 2;
InsertMenuItem (hMenu2, 0, FALSE, &mii);
```

```
lstrcpy (string, "Datei");
mii.fMask = MIIM_TYPE | MIIM_SUBMENU;
mii.hSubMenu = hMenu2;
InsertMenuItem (hMenu, 0, FALSE, &mii);
```

```
HMENU hMenu2_;
hMenu2_ = CreatePopupMenu ();
lstrcpy (string, "Autor");
mii.fMask = MIIM_TYPE | MIIM_ID;
mii.wID = 3;
InsertMenuItem (hMenu2_, 0, FALSE, &mii);
```

```
HMENU hMenu3;
hMenu3 = CreatePopupMenu ();
lstrcpy (string, "Farbtiefe");
mii.fMask = MIIM_TYPE | MIIM_ID;
mii.wID = 4;
InsertMenuItem (hMenu3, 0, FALSE, &mii);
lstrcpy (string, "Aufloesung");
mii.wID = 5;
InsertMenuItem (hMenu3, 0, FALSE, &mii);
```

```
HMENU hMenu3_;
hMenu3_ = CreatePopupMenu ();
lstrcpy (string, "Windows Verzeichniss");
mii.wID = 6;
```



```
InsertMenuItem (hMenu3_, 0, FALSE, &mii);  
lstrcpy (string, "Aktuelles Verzeichniss");  
mii.wID = 8;  
InsertMenuItem (hMenu3_, 0, FALSE, &mii);
```

```
lstrcpy (string, "Grafik");  
mii.fMask = MIIM_TYPE | MIIM_SUBMENU;  
mii.hSubMenu = hMenu3;  
InsertMenuItem (hMenu2_, 0, FALSE, &mii);  
lstrcpy (string, "Festplatte");  
mii.hSubMenu = hMenu3_;  
InsertMenuItem (hMenu2_, 0, FALSE, &mii);
```

```
lstrcpy (string, "Information");  
mii.hSubMenu = hMenu2_  
InsertMenuItem (hMenu, 0, FALSE, &mii);
```

```
SetMenu (hWnd, hMenu);
```

```
delete string[];
```

### *Das oberste Menü in der Hierarchie*

Eine Variable vom Typ `HMENU` wird angelegt. Dies ist ein Handle auf ein Menüobjekt. Dann wird das Menüobjekt erstellt und der Handle zu dem Objekt der Variablen übergeben. Dies geschieht mit der Funktion `CreateMenu`.

```
HMENU CreateMenu(VOID);
```

Im Quelltext hat die Funktion fast genau dasselbe Erscheinungsbild.

```
hMenu = CreateMenu ();
```

- Die Funktion hat keine Parameter. Es wird nur ein Handle auf ein Menüobjekt zurückgeliefert.

### *Einträge in das Menü setzen*

Um Einträge in das Menüobjekt zu setzen, benutzt man die Funktion `InsertMenuItem`.





```

BOOL WINAPI InsertMenuItem( HMENU hMenu, UINT uItem,
                            BOOL fByPosition,
                            LPMENITEMINFO lpmi);

```

Die Funktion wird im Quelltext mehrfach verwendet.

```

InsertMenuItem(hMenu, 0, FALSE, &mii);
InsertMenuItem(hMenu2, 0, FALSE, &mii);
InsertMenuItem(hMenu, 0, FALSE, &mii);
InsertMenuItem(hMenu2_, 0, FALSE, &mii);
InsertMenuItem(hMenu3, 0, FALSE, &mii);
InsertMenuItem(hMenu3, 0, FALSE, &mii);
InsertMenuItem(hMenu3_, 0, FALSE, &mii);
InsertMenuItem(hMenu3_, 0, FALSE, &mii);
InsertMenuItem(hMenu2_, 0, FALSE, &mii);
InsertMenuItem(hMenu2_, 0, FALSE, &mii);
InsertMenuItem(hMenu, 0, FALSE, &mii);

```

- `hMenu` ist der Handle auf das Menüobjekt, in das der Eintrag geschrieben werden soll.
- `uItem` ist entweder eine Identifikationsnummer für den Menüeintrag oder eine Position, an der der Menüeintrag erscheinen soll. Ob der Wert zur Identifikation oder zur Angabe einer Position dient, wird durch den nächsten Parameter entschieden. Dieser Wert wird auf `FALSE` gesetzt, weil er nicht benötigt wird.
- `fByPosition` ist vom Typ `BOOL` und kann somit nur den Wert `TRUE` oder `FALSE` haben. Bei `FALSE` ist der Wert `uItem` ein Identifizierer, bei `TRUE` ist er ein Wert, der die Position des Menüeintrages angibt. Dieser Wert wird auf `0` gesetzt, weil er nicht benötigt wird.
- Als Letztes folgt ein Zeiger auf eine Struktur vom Typ `MENITEMINFO`. Diese Struktur enthält weitere Informationen, die dem Menüeintrag mitgegeben werden.

Die Struktur `MENITEMINFO` wird nicht nur von dem Befehl `InsertMenuItem` benutzt, sondern auch von anderen Befehlen. Sie kann zum Setzen und zum Anfordern von Eigenschaften eines Menüeintrages verwendet werden.





```
typedef struct tagMENUITEMINFO
{
    UINT    cbSize;
    UINT    fMask;
    UINT    fType;
    UINT    fState;
    UINT    wID;
    HMENU   hSubMenu;
    HBITMAP hbmpChecked;
    HBITMAP hbmpUnchecked;
    DWORD   dwItemData;
    LPTSTR  dwTypeData;
    UINT    cch;
} MENUITEMINFO, FAR *LPMENUITEMINFO;
```

- `cbSize` ist die Größe der Struktur in Bytes.
- `fMask` setzt sich aus mehreren Konstanten zusammen und gibt an, welche Daten geholt oder gesetzt werden sollen.

`MIIM_ID` bedeutet, dass ein ID für das Menü-Item gesetzt, oder dass der Wert des Menüeintrages geholt werden soll.

`MIIM_TYPE` bedeutet, dass der Wert von `fType` und `dwTypeData` gesetzt oder geholt werden soll.

`MIIM_SUBMENU` bedeutet, dass der Wert von `hSubMenu` gesetzt oder geholt wird.

- `fType` gibt Informationen darüber, welche Eigenschaft der Menüeintrag hat. Der Menüeintrag kann z.B. ein Text-String, eine Trennlinie oder ein Bitmap sein.

`MFT_STRING` bedeutet, dass der Menüeintrag ein Text-String ist.

- `fState` gibt Aufschluss über den Status von Menüeinträgen. Dies ist nur erforderlich, wenn der Menüeintrag zum Beispiel eine Check-Box besitzt, oder wenn er nicht angeklickt werden darf.
- `wID` ist ein Wert, der den Identifizierer des Menüeintrages repräsentiert.

- `hSubMenu` ist der Handle auf ein Menüobjekt. Dieser Handle wird mit dem Menüeintrag verbunden. Falls der Menüeintrag auf ein Untermenüobjekt verweist, muss dieser Wert gesetzt werden.
- `hBmpChecked` ist der Handle auf ein Bitmap-Objekt, das für den Menüeintrag angezeigt werden soll. Dieses Bitmap-Objekt muss in diesem Fall angekreuzt sein.
- `hBmpUnChecked` ist der Handle auf das Bitmap, wenn es nicht angekreuzt ist.
- `dwItemData` ist ein von der Anwendung definierter Wert.
- `dwTypeData` enthält einen Zeiger auf die Daten, die durch `fType` gesetzt worden sind.
- `cch` gibt die Länge des Menü-Item-Texts an, falls diese erforderlich ist.

### *Das Menü dem Fenster zuweisen*

Nachdem die gesamte Hierarchie des Menüs erstellt worden ist, wird das Menü nun mit dem Befehl `SetMenu` dem Fenster zugewiesen.

```
BOOL SetMenu( HWND hWnd, HMENU hMenu);
```

Die Funktion wird der Menüerstellung am Ende zugewiesen.

```
SetMenu (hWnd, hMenu);
```

- `hWnd` ist der Handle auf das Fensterobjekt, welches das Menü erhalten soll.
- `hMenu` ist der Handle auf das Menüobjekt, das dem Fensterobjekt zugewiesen wird.

### **Ausführen von Menü Aktionen**

Hierzu benutzt man die Nachricht `WM_COMMAND`. Wenn ein Menüpunkt angeklickt wurde, wird `WM_COMMAND` in die Warteschlange der Anwendung eingefügt.

```
WM_COMMAND
wNotifyCode = HIWORD(wParam);
wID = LOWORD(wParam);
hwndCtl = (HWND) lParam;
```





- `wNotifyCode` hat bei einer Nachricht von einem Menüpunkt immer den Wert 0.
- `wID` ist der ID-Wert des Menüpunkts. Er entspricht dem Wert, der in der Struktur `MENUITEMINFO` übergeben wurde.
- `hwndCtl` ist bei Menünachrichten immer `NULL`.

Aufgrund dieser Daten erfolgt dann die Auswertung in der Fensterfunktion. Anschließend starten einige Aktionen, von denen wir jetzt die neuen Hauptbefehle näher betrachten wollen.

### *Gerätekontext Informationen*

Mit der Funktion `GetDeviceCaps` bekommt man Informationen über die Gerätekontexte, also über bestimmte Geräte des Systems. Zu diesen Geräten zählen z.B. Grafikkarte, Drucker oder Kamera.

```
int GetDeviceCaps( HDC hdc, int nIndex);
```

Die Funktion tritt im Quelltext einige Male auf:

```
BitsPixel = GetDeviceCaps (hdc, BITSPIXEL);
```

```
HRes = GetDeviceCaps (hdc, HORZRES);
```

```
VRes = GetDeviceCaps (hdc, VERTRES);
```

- `hdc` ist der Handle auf das Gerätekontext-Objekt. Dieses Gerätekontext-Objekt ist mit einem Gerät verknüpft. Über dieses Gerät werden die Informationen eingeholt.
- `nIndex` verlangt eine Konstante, die besagt, welcher Wert zurückgeliefert werden soll.
- Als `return` bekommt man den Wert, der die Information über das Gerät enthält.

### *Verzeichnisse ermitteln*

Es gibt zwei Arten von Verzeichnissen, die man standardmäßig ermitteln kann. Dazu gehören das Windowsverzeichnis, das Systemverzeichnis und das aktuelle Verzeichnis. Das Windowsverzeichnis ist das Verzeichnis, das bei der Installation von Windows festgelegt wurde. Es enthält Windowsanwendungen, Initialisierungsdateien und Hilfedateien. Das Systemverzeichnis enthält DLL-, Treiber- und Font-Dateien. Es ist standardmäßig unter dem Windowsverzeichnis mit dem Namen `System` festgelegt. Das aktuelle Verzeichnis ist für

jede Anwendung verschieden und ist das Verzeichnis, aus dem die Anwendung gestartet wurde, es sei denn, es wurde danach verändert.

Die Funktion `GetWindowsDirectory` ermittelt das Windows-Verzeichnis.

```
UINT GetWindowsDirectory( LPTSTR lpBuffer, UINT uSize);
```

Die Funktion wird im Quelltext einmal verwendet.

```
GetWindowsDirectory (string2, 500);
```

- `lpBuffer` ist ein Zeiger vom Typ `char`.
- `uSize` ist die Größe des Arrays auf den `lpBuffer` zeigt.

Die Funktion `GetCurrentDirectory` ermittelt das aktuelle Verzeichnis.

```
DWORD GetCurrentDirectory( DWORD nBufferLength,  
                          LPTSTR lpBuffer);
```

Sie kommt im Quelltext einmal vor:

```
GetCurrentDirectory (500, string2);
```

- `nBufferLength` ist die Größe des Arrays, auf den der nächste Parameter zeigt.
- `lpBuffer` ist ein Zeiger vom Typ `char`.







# Behandlung von Dateien

## 9.1 Allgemeines

### 9.1.1 Dateisysteme

Es gibt mehrere Dateisysteme, die von verschiedenen Windowsversionen unterstützt werden. So unterstützt Windows NT das NTFS und Windows 95/98 das „Protected FAT File System“. Diese Dateisysteme sind Definitionen, wie Dateien und Verzeichnisse auf der Festplatte abgespeichert werden. Dazu gehören z. B. Informationen, wie lang Dateinamen sein, welche Zeichen darin vorkommen dürfen oder welche Sicherheitsattribute die Dateien tragen.

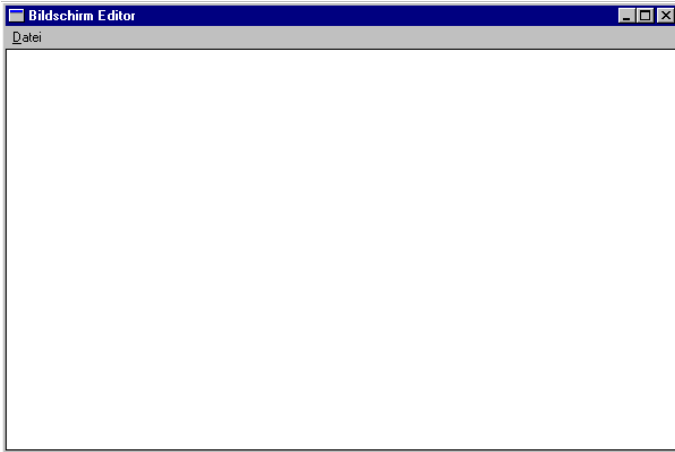
### 9.1.2 Funktionen

Windows stellt für diese Dateisysteme Funktionen bereit. Diese Funktionen laufen unter allen Dateisystemen. Deshalb kann der Code, der für ein Programm für Windows 95 geschrieben wurde, ohne Probleme auf Windows NT ausgeführt werden. Auch die Dateiverwaltung funktioniert dort entsprechend.

## 9.2 Die Anwendung

### 9.2.1 Der Quelltext

Die folgende Anwendung ist ein Editor, der nur auf den Bildschirm bezogen ist. Er besitzt ein Menü. Mit dem Menüpunkt „Beenden“ wird der Editor beendet. Die Funktion `Neu` setzt den Zeichenpuffer auf 0 Zeichen. Es kann also neuer Text eingegeben werden. Die Funktion „Speichern“, speichert den Text aus dem Edit-Feld, welches den Hauptbestandteil des Editors ausmacht, in der Datei `c:\TEXT.TXT`. Mit der Funktion „Öffnen“ wird die vorhandene Datei `c:\TEXT.TXT` geöffnet. Damit wären die wichtigsten Dateifunktionen in das Programm eingebunden.



*Bild 9.1: Anwendung nach dem Start*



*Bild 9.2: Anwendung mit einem Textbeispiel*

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```



```
HINSTANCE hInstGlobal;
HWND hEdit;

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    hInstGlobal = hInstance;

    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
    hWindow = CreateWindow("WinProg", "Bildschirm Editor",
                        WS_OVERLAPPEDWINDOW,
                        0, 0, 600, 400, NULL, NULL,
                        hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
```

```

{
    TranslateMessage (&Message);
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_CREATE:
            HMENU hMenu;
            hMenu = CreateMenu ();
            MENUITEMINFO mii;
            mii.cbSize = sizeof(MENUITEMINFO);
            mii.fMask = MIIM_TYPE | MIIM_ID;
            mii.fType = MFT_STRING;
            char *string;
            string = new char[40];
            mii.dwTypeData = string;
            mii.cch = lstrlen (string);

            HMENU hMenu2;
            hMenu2 = CreatePopupMenu ();

            lstrcpy (string, "&Neu");
            mii.dwTypeData = string;
            mii.cch = lstrlen (string);
            mii.wID = 1;
            InsertMenuItem (hMenu2, 1, TRUE, &mii);

            lstrcpy (string, "&0effnen c:\\text.txt");
            mii.dwTypeData = string;
            mii.cch = lstrlen (string);
            mii.wID = 2;
    }
}

```

```
InsertMenuItem (hMenu2, 2, TRUE, &mii);

lstrcpy (string, "&Speicher c:\\text.txt");
mii.dwTypeData = string;
mii.cch = lstrlen (string);
mii.wID = 3;
InsertMenuItem (hMenu2, 3, TRUE, &mii);

mii.fMask = MIIM_TYPE;
mii.fType = MFT_SEPARATOR;
InsertMenuItem (hMenu2, 4, TRUE, &mii);

mii.fMask = MIIM_TYPE | MIIM_ID;
mii.fType = MFT_STRING;
lstrcpy (string, "&Beenden");
mii.dwTypeData = string;
mii.cch = lstrlen (string);
mii.wID = 4;
InsertMenuItem (hMenu2, 5, TRUE, &mii);

lstrcpy (string, "&Datei");
mii.dwTypeData = string;
mii.cch = lstrlen (string);
mii.fMask = MIIM_TYPE | MIIM_SUBMENU;
mii.hSubMenu = hMenu2;
InsertMenuItem (hMenu, 1, FALSE, &mii);

SetMenu (hWnd, hMenu);

RECT rect;
GetClientRect (hWnd, &rect);
hEdit = CreateWindow("EDIT", "",
                    WS_CHILD | WS_VISIBLE |
                    WS_BORDER | ES_MULTILINE,
                    0,0,rect.right,rect.bottom,
                    hWnd,(HMENU) 1,
                    hInstGlobal, NULL);

return 0;
```



```
case WM_SIZE:
    GetClientRect (hWnd, &rect);
    MoveWindow (hEdit, 0, 0, rect.right,
                rect.bottom,TRUE);
    return 0;
case WM_COMMAND:
    if (HIWORD(wParam) == 0)
    {
        switch LOWORD(wParam)
        {
            case 1:
                char *EditString;
                EditString = new char[80];
                lstrcpy (EditString, "");
                SetWindowText (hEdit, EditString);
                delete [] Editstring;
                return 0;
            case 2:
                HANDLE hFile;
                hFile = CreateFile ("c:\\text.txt",
                                    GENERIC_READ,0,
                                    NULL,OPEN_EXISTING,
                                    NULL,NULL);

                DWORD Size;
                Size = GetFileSize (hFile, NULL);
                char *FileText;
                FileText = new char[Size+1];
                DWORD Readd;
                ReadFile (hFile,FileText,Size+1,
                          &Readd,NULL);
                CloseHandle (hFile);
                SetWindowText (hEdit, FileText);
                delete [] Editstring;
                return 0;
            case 3:
                hFile = CreateFile ("c:\\text.txt",
                                    GENERIC_WRITE,0,NULL,
                                    CREATE_ALWAYS,
```

```
        NULL,NULL);
    FileText = new char[GetWindowTextLength(
        hEdit)+1];
    GetWindowText (hEdit, FileText,
        GetWindowTextLength(
            hEdit)+1);
    WriteFile (hFile,FileText,
        GetWindowTextLength(hEdit)+1,
        &Readd,NULL);
    CloseHandle (hFile);
    delete [] FileText;
    return 0;
case 4:
    PostQuitMessage (0);
    return 0;
}
}
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
        wParam, lParam);
}
}
```

### 9.2.2 Erläuterung

Da die Anwendung zum größten Teil bekannte Elemente enthält, konzentrieren wir uns auf die Dateifunktionen.

#### Eine Datei öffnen oder neu erstellen

Wenn man die Funktion DATEI/ÖFFNEN im Menü anklickt, wird entweder eine bestehende Datei geöffnet oder eine neue Datei angelegt. Die Daten dieser Datei werden in einen Buffer eingelesen. Zu diesem Zweck wird ein Dateiojekt erstellt, das dann wieder geschlossen werden muss, damit auch andere Programme auf diese

Datei zugreifen können. Zum Schluss werden die eingelesenen Daten an das Edit-Feld übergeben.

case 2:

```
HANDLE hFile;
hFile = CreateFile ("c:\\text.txt",
                   GENERIC_READ,0,
                   NULL,OPEN_EXISTING,
                   NULL,NULL);

DWORD Size;
Size = GetFileSize (hFile, NULL);
char *FileText;
FileText = new char[Size+1];
DWORD Readd;
ReadFile (hFile,FileText,Size+1,
          &Readd,NULL);

CloseHandle (hFile);
SetWindowText (hEdit, FileText);
return 0;
```

### *CreateFile*

Die Funktion `CreateFile` erstellt ein Dateiojekt und liefert einen Handle auf dieses Dateiojekt zurück.

```
HANDLE CreateFile( LPCTSTR lpFileName,
                  DWORD dwDesiredAccess,
                  DWORD dwShareMode,
                  LPSECURITY_ATTRIBUTES
                  lpSecurityAttributes,
                  DWORD dwCreationDisposition,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile);
```

Die Funktion wird im Quelltext einerseits zum Öffnen oder Neuerstellen einer Datei verwendet, andererseits wird sie aber auch dazu genutzt, eine Datei zum Speichern zu erstellen.

```
hFile = CreateFile ("c:\\text.txt",
                   GENERIC_READ,0,
                   NULL,OPEN_EXISTING,
                   NULL,NULL);
```

```
hFile = CreateFile ("c:\\text.txt",
                  GENERIC_WRITE, 0, NULL,
                  CREATE_ALWAYS,
                  NULL, NULL);
```

- `lpFileName` ist ein Zeiger auf einen Array des Typs `char`. Hier müssen Name und Pfad der Datei angegeben werden.
- `dwDesiredAccess` bestimmt die Art des Zugriffs auf das jeweilige Objekt. Zwei Arten werden bei einem Dateizugriff benötigt:  
`GENERIC_READ` bestimmt die Datei für Lesezugriff.  
`GENERIC_WRITE` bestimmt die Datei für Schreibzugriff.
- `DwSharedMode` bestimmt, ob man auf das Objekt gleichzeitig mehrmals zugreifen kann. Dieser Wert wird auf `0` gesetzt, damit dies nicht möglich ist.
- `LpSecurityAttributes` ist ein Zeiger auf eine Struktur `SECURITY_ATTRIBUTES`. Diese Struktur enthält Angaben über die Vererbung dieses Objekts und über seine Zugriffsrechte unter Windows NT. Dieser Wert wird auf `NULL` gesetzt. Das bedeutet, dass das Objekt nicht vererbt werden kann.
- `DwCreationDisposition` gibt weitere Informationen für die Aktion, die mit `dwDesiredAccess` gesetzt wurde.  
`OPEN_EXISTING` bedeutet, dass nur existierende Dateien für einen Zugriff geöffnet werden sollen.  
`CREATE_ALWAYS` bestimmt, dass auch eine neue Datei erstellt wird, wenn die Datei schon vorhanden ist.
- `DwFlagsAndAttributes` belegt die Datei Attribute für Benutzerzugriff und Server/Client Zugriff. Dieser Wert wird auf `NULL` gesetzt.
- `HTemplateFile` wird auf `NULL` gesetzt. Auf die Bedeutung dieses Parameters gehen wir hier nicht näher ein.

### ReadFile

Die Funktion `ReadFile` liest Daten aus einer Datei in einen Buffer ein.

```
BOOL ReadFile( HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
```

```
LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped);
```

Im Quelltext wird diese Funktion einmal benutzt, um eine Datei zu lesen.

```
ReadFile (hFile,FileText,Size+1,
          &Readd,NULL);
```

- `HFile` ist der Handle auf das Dateiobjekt.
- `LpBuffer` ist ein Zeiger auf einen Buffer.
- `NNumberOfBytesToRead` ist die Anzahl der Bytes, die aus der Datei gelesen werden sollen.
- `LpNumerOfBytesRead` ist die Anzahl der gelesenen Bytes. Da dieser Wert zurück gegeben wird, muss er als Zeiger angegeben werden.
- `LpOverlapped` ist dafür zuständig, dass eine Datei ab einem bestimmten Offset gelesen wird. Dieser Wert wird auf NULL gesetzt. So wird die Datei immer wieder vom Anfang gelesen.

### *WriteFile*

Die Funktion `WriteFile` schreibt Daten aus einem Buffer in ein Dateiobjekt.

```
BOOL WriteFile( HANDLE hFile,
                LPCVOID lpBuffer,
                DWORD nNumberOfBytesToWrite
                LPDWORD lpNumberOfBytesWritten,
                LPOVERLAPPED lpOverlapped);
```

Die Funktion ist einmal im Quelltext vorhanden. Jede Funktion mit Dateizugriff ändert die Daten meistens direkt in der Datei.

```
WriteFile (hFile,FileText,
          GetWindowTextLength(hEdit)+1,
          &Readd,NULL);
```

- `HFile` ist der Handle auf das Dateiobjekt, in das die Daten aus dem Buffer übertragen werden sollen.
- `LpBuffer` ist ein Zeiger auf den Buffer.
- `NNumberOfBytesToWrite` ist die Anzahl der Bytes, die aus dem Buffer in das Dateiobjekt übertragen werden sollen.



- `LpNumberOfBytesWritten` gibt an, wie viele Bytes wirklich in das Dateiojekt geschrieben wurden.
- `LpOverlapped` gibt den Offset an. Dieser Wert hat die gleiche Bedeutung wie der von `ReadFile`.

### *CloseHandle*

Mit der Funktion `CloseHandle` wird ein Dateiojekt entfernt.

```
BOOL CloseHandle( HANDLE hObject);
```

Die Funktion wird einmal zum Schließen des Dateiojekts für Lesezugriffe und einmal für Schreibzugriffe aufgerufen.

```
CloseHandle (hFile);
```

- `hFile` ist der Handle auf das Dateiojekt, das geschlossen werden soll.





# Anwendungen, Prozesse und Threads

## 10.1 Allgemeines

Eine *Anwendung* ist die EXE-Datei auf der Festplatte. Sie enthält alle Daten.

Ein *Prozess* ist die Anwendung, wenn sie ausgeführt wird. So können mehrere Prozesse derselben Anwendung auf einem System gleichzeitig laufen.

Für die Ausführung des Codes in der Anwendung sorgen *Threads*. Sie sagen dem System, wo der nächste Befehl zur Ausführung ansteht. Man kann für einen Prozess mehrere Threads kreieren. Dieses Verfahren nennt man *Multi-Threading*. Man benutzt es, wenn man z.B. eine sehr zeitaufwendige Datenstruktur bearbeitet. Während diese Datenstruktur bearbeitet wird, muss der Benutzer das Fenster noch verschieben und ggf. die Verarbeitung der Daten abbrechen können. Dazu verwendet man dann einen gesonderten Thread, der nur die Datenverarbeitung übernimmt. Sehr wichtig ist dabei, dass jeder Thread seine eigene Warteschlange für Nachrichten hat. Das heißt die bisherigen Erläuterungen der Anwendungen waren stark vereinfacht, um sich zunächst auf das Wesentlichste konzentrieren zu können.

## 10.2 Eine Multi-Threading-Anwendung

### 10.2.1 Der Quelltext

Die folgende Anwendung erstellt ein Fenster. Die Nachrichtenverarbeitung soll noch weiterlaufen. Nebenbei soll ein Vorgang stattfinden, der sehr zeitaufwendig ist. Es wird ein Thread erstellt, der diesen Vorgang übernimmt. Bei diesem Vorgang handelt es sich um eine Schleife, die immer wieder neue Rechtecke zeichnet. Während dieses Vorgangs kann das Fenster bewegt oder auch verkleinert werden.

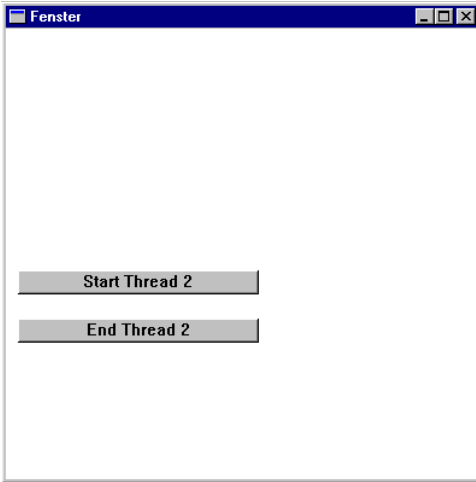


Bild 10.1: Die Anwendung nach dem Start

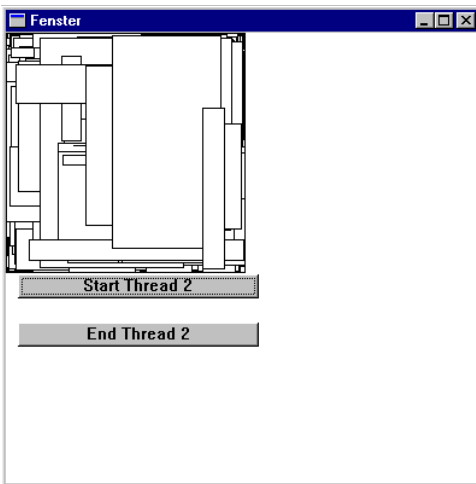


Bild 10.2: Die Anwendung nach dem Start von Thread 2

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
DWORD WINAPI ThreadProc ( LPVOID);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow,hButton;

    hWindow = CreateWindow("WinProg","Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0,0,400,400,NULL,NULL,
                          hInstance, NULL);

    hButton = CreateWindow("BUTTON","Start Thread 2",
                          WS_CHILD | WS_VISIBLE
                          | BS_PUSHBUTTON,
                          10,200,200,20,hWindow,
                          (HMENU) 1, hInstance, NULL);
```

```
hButton = CreateWindow("BUTTON", "End Thread 2",
    WS_CHILD | WS_VISIBLE
    | BS_PUSHBUTTON,
    10, 240, 200, 20, hWindow,
    (HMENU) 2, hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

DWORD WINAPI ThreadProc( LPVOID pvoid )
{
    int wert = 2;
    while (wert == 2)
    {
        HDC hdc1;
        hdc1 = GetDC (hWindow);
        Rectangle (hdc1, rand()%200, rand()%200 ,
            rand()%200, rand()%200);
        ReleaseDC (hWindow, hdc1);
    }
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
    WPARAM wParam, LPARAM lParam)
{
    static HANDLE hThread;
```

```
switch(uiMessage)

case WM_COMMAND:
    if (HIWORD(wParam) == BN_CLICKED)
    {
        if (LOWORD(wParam) == 1)
        {
            DWORD dwThreadParam = 1;
            DWORD dwThreadID;
            hThread = CreateThread (NULL, 0, ThreadProc,
                                   &dwThreadParam, 0,
                                   &dwThreadID);
        }
        if (LOWORD(wParam) == 2)
        {
            TerminateThread (hThread, 0);
        }
    }
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}
```

### 10.2.2 Erläuterung

Dieses Programm erstellt den neuen Thread, der in das Fenster zeichnet, während der erste Thread des Prozesses weiterhin die Nachrichtenverarbeitung übernimmt. Jeder Thread hat seine eigene Warteschlange zur Nachrichtenverarbeitung, weswegen alle Nachrichten für die schon erstellten Fenster der Nachrichtenwarteschlange des anderen Threads zugeordnet werden.

Die Funktion `CreateThread` erstellt den neuen Thread und weist ihm eine Funktion zu.

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES
                    lpThreadAttributes,
                    DWORD dwStackSize,
                    LPTHREAD_START_ROUTINE
                    lpStartAddress,
                    LPVOID lpParameter,
                    DWORD dwCreationFlags,
                    LPDWORD lpThreadId);
```

Die Funktion wird im Quelltext folgendermaßen verwendet.

```
hThread = CreateThread (NULL, 0, ThreadProc,
                       &dwThreadParam, 0,
                       &dwThreadId);
```

- `lpThreadAttributes` bilden eine Struktur vom Typ `SECURITY_ATTRIBUTES`, die bestimmte Sicherheitsbestimmungen für den Thread festlegt. Dieser Wert wird einfach auf `NULL` gesetzt, damit der Handle nicht vererbt werden kann.
- `DwStackSize` ist die Größe des Stacks bei der Initialisierung. Bei `NULL` wird die gleiche Größe wie bei dem aufrufenden Thread verwendet.
- `lpStartAddress` ist ein Zeiger auf den Anfang der Funktion, die der Thread ausführt.
- `lpParameter` ist ein Wert mit 32 Bit, der dem Thread mitgeteilt wird. Dieser Wert wird der Funktion übergeben, die der Thread ausführt. Er muss als Zeiger übergeben werden.
- `DwCreationFlags` ist ein Wert, der durch bestimmte Konstanten festgelegt wird. Der Thread kann so initialisiert werden, dass er nicht ausgeführt wird (*suspended*), oder aber mit einem Wert, der angibt, dass der Thread ausgeführt werden soll, sobald er erstellt wurde.
- `lpThreadId` ist ein Zeiger auf einen 32-Bit-Wert, der einen Bezeichner für den Thread enthält.
- Als `return` liefert diese Funktion den Wert des Handles auf das Thread-Objekt.



Die Funktion `TerminateThread` löscht den Thread, so dass der Code der Funktion nicht mehr ausgeführt wird.

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode);
```

Die Funktion wird aufgerufen, wenn der Button mit der Kennziffer 2 gedrückt wurde.

```
TerminateThread (hThread, 0);
```

- `hThread` ist der Handle auf das Thread-Objekt, welches gelöscht werden soll.
- `DwExitCode` ist der Exit-Code für den Thread.



# // DLL-Dateien

## 11.1 Allgemeines

Man kann in C++ Library-Dateien mit dem gesamten Code einbinden (z.B. `Stdlib.lib`). Das bedeutet: jedes Programm hat sämtlichen Code der Library-Dateien gespeichert. Um dieser Sache vorzubeugen, hat Microsoft die DLL-Dateien erfunden (*Dynamic Link Library*). Man bindet nach wie vor die Header-Dateien und Library-Dateien in das Programm ein, aber der Code der einzelnen Funktionen steht nicht mehr in den Library-Dateien, sondern in den DLL-Dateien. Es werden nur noch Verweise auf den Code in den DLL-Dateien in das Programm eingebunden. Und es muss nur eine DLL-Datei im Speicher sein, die von allen Programmen verwendet wird. Falls die entsprechende Datei noch nicht im Speicher ist, wird sie geladen. Es gibt auch die Möglichkeit, DLL-Dateien direkt aus dem Quellcode des Programmes zu laden. DLL-Dateien haben mehrere Vorteile. Zum einen sparen sie Speicher und zum anderen lassen sich Versionen einfach durch den Austausch von DLL-Dateien updaten.

Die gesamte Win32-API wurde als eine Sammlung von DLL-Dateien entworfen.

In einer DLL-Datei gibt es Funktionen und Daten. Diese Funktionen und Daten sind intern oder extern. Intern bedeutet, dass die Funktionen und Daten nur von Funktionen der DLL-Datei genutzt werden. Extern bedeutet, dass die Funktionen und Daten von einem Programm verwendet werden.

DLL-Dateien werden auf verschiedene Weise genutzt: z.B. als „load-time-dynamic-link-Dateien“. Diese Dateien werden durch den Start des Programmes geladen und befinden sich dann im virtuellen Adressraum des Prozesses. Der Code des Prozesses wird so verändert, dass die DLL-Funktionen aufgerufen werden. Die Veränderung wird mit den Informationen aus der Library-Datei ausgeführt. Sie

werden aber auch als „run-time-dynamic-link“ genutzt. Eine DLL-Datei wird mit `LoadLibrary` geladen. Durch die Funktion `GetProcAddress` werden die Adressen der Funktionen der DLL-Datei herausgefunden. Diese werden dazu genutzt, die Funktionen aufzurufen.

In diesem Kapitel werden wir ausschließlich die load-time-dynamic-link-Dateien besprechen.

## 11.2 Die Anwendung

### 11.2.1 Der Quellcode der DLL-Datei

Die folgende DLL-Datei stellt eine Funktion bereit. Diese Funktion macht nichts anderes, als in einen übergebenen Gerätekontext-Handle ein Rechteck zu zeichnen. Beim Erstellen einer DLL-Datei wird eine LIB-Datei angelegt. Diese LIB-Datei enthält Verweise auf die Funktionen in der DLL-Datei. Die Header-Datei dagegen muss man selber schreiben. Sie ist für den Erstellungsprozess gedacht.

```
#include <windows.h>
#include "Rechteck.h"

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason,
                  PVOID pvReserved)
{
    return TRUE;
}

__declspec ( dllexport ) BOOL CALLBACK Rechteck (HDC hdc)
{
    Rectangle (hdc, 10, 10, 200, 200);
    return 0;
}
```

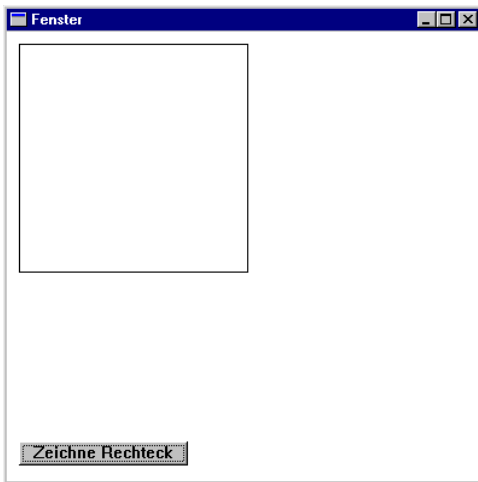
Die Header-Datei sieht folgendermaßen aus.

```
__declspec ( dllexport ) BOOL CALLBACK Rechteck (HDC);
```

Nachdem man die DLL-Datei erstellt hat, muss man die Header- und die LIB-Datei in den Quellcode einbinden.



*Bild 11.1: Die Anwendung nach dem Start*



*Bild 11.2: Die Anwendung nach der Nutzung durch die DLL-Datei*

//

Das eigentliche Programm hat folgenden Quelltext:

```
#include <windows.h>
#include "Rechteck.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow,hButton;
    hWindow = CreateWindow("WinProg","Fenster",
                        WS_OVERLAPPEDWINDOW,
                        0,0,400,400,NULL,NULL,
                        hInstance, NULL);
    hButton = CreateWindow("BUTTON","Zeichne Rechteck",
                        WS_CHILD | WS_VISIBLE
                        | BS_PUSHBUTTON,
                        10,340,140,20,hWindow,(HMENU) 1,
                        hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);
```

```

UpdateWindow (hWindow);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_COMMAND:
            if (HIWORD(wParam) == BN_CLICKED)
            {
                if (LOWORD(wParam) == 1)
                {
                    HDC hdc;
                    hdc = GetDC (hWnd);
                    Rechteck (hdc);
                    ReleaseDC (hWnd, hdc);
                }
            }
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default:
            return DefWindowProc (hWnd, uiMessage,
                                   wParam, lParam);
    }
}

```

//

## 11.2.2 Erläuterung

### Eine DLL-Datei erstellen

Für die Erstellung einer DLL-Datei gelten die gleichen Prinzipien wie für eine LIB-Datei. Man erstellt wie gewohnt seine Funktionen. Auf diese Funktionen verweist man dann aus der Header-Datei. Anschließend entwickelt man mit einem Programm die DLL-Datei und eine LIB-Datei. Die LIB-Datei enthält aber diesmal nur Verweise auf die Funktionen in der DLL-Datei. Später ruft man die Funktionen so auf, als hätte man eine normale LIB-Datei erstellt.

#### *Die Funktion DllMain*

Die Funktion `DllMain` wird aufgerufen, wenn ein Programm die Datei neu initialisiert oder die nicht mehr benötigt. Sie ist optional und liefert drei Parameter.

```
int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason,
                  PVOID pvReserved)
```

- `HInstance` ist ein Handle auf die DLL. Der Wert des Handles ist die Basisadresse der DLL-Datei im Speicher.
- `FdwReason` ist ein Wert, der bestimmt, ob die DLL-Datei in den virtuellen Adressraum geladen wurde, oder ob sie aus dem virtuellen Adressraum herausgenommen werden soll. Diese beiden Möglichkeiten bezeichnen die Konstanten `DLL_PROZESS_ATTACH` (laden) und `DLL_PROZESS_DETACH` (herausnehmen). Es gibt auch noch andere Situationen, in denen die Funktion `DllMain` aufgerufen wird.

#### *Eine ganz normale Funktion für eine DLL-Datei*

Funktionen werden fast genauso definiert, wie in einer LIB-Datei. Das sorgt dafür, dass die Funktionen nach außen sichtbar sind, und dass auch andere Sprachen sie aufrufen können.

Diese Funktion zeichnet auf einen übergebenen Gerätekontext-Handle ein Rechteck. Dafür braucht man natürlich keine DLL-Datei.



```
__declspec ( dllexport ) BOOL CALLBACK Rechteck (HDC hdc)
{
    Rectangle (hdc, 10, 10, 200, 200);
    return 0;
}
```

### *Die Header-Datei*

In der Header-Datei wird der Funktionsname - wie üblich - noch einmal erwähnt.

```
__declspec ( dllexport ) BOOL CALLBACK Rechteck (HDC);
```

### **Das Programm zum Aufruf der DLL-Datei**

Eine DLL-LIB-Datei wird wie eine normale LIB-Datei aufgerufen. Es wird nur dafür gesorgt, dass die DLL-Datei in den Speicher geladen wird. Man muss die Header-Datei einbinden und mehr wäre zu dem Programm auch nicht zu sagen. Man kann die Funktion jetzt wie gewohnt verwenden.

//



## 12.1 Allgemeines

Ein Timer ruft eine Funktion in bestimmten Zeitabständen auf. Er kann aber auch in bestimmten Zeitabständen eine Nachricht WM\_TIMER an ein Fenster senden. Timer werden vom System verwaltet.

## 12.2 Eine Timer-Anwendung mit Timer-Nachrichten

### 12.2.1 Der Quelltext

Diese Anwendung erstellt einen Timer, der in bestimmten Zeitabständen Nachrichten an das Fenster sendet. Zwei Zähler  $X_{pos}$  und  $Y_{pos}$  werden vereinbart. Durch diese beiden Zähler werden Position und Größe des Fensters neu bestimmt. Das Fenster wird ständig seine Position und seine Größe verändern.

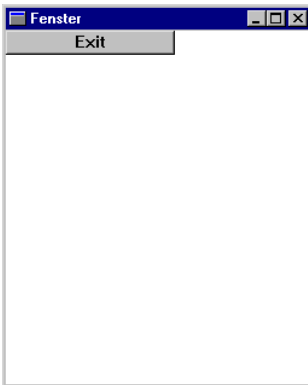


Bild 12.1: Die Anwendung eine gewisse Zeit nach dem Start



Bild 12.2: Die Anwendung nach einiger Zeit

```
#include <windows.h>

int XPos, YPos;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    XPos = 0;
    YPos = 0;
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
```

```

WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hWindow,hButton,hButtonExit;
hWindow = CreateWindow("WinProg","Fenster",
    WS_OVERLAPPEDWINDOW,
    XPos,YPos,
    400-XPos,400-YPos,NULL,NULL,
    hInstance, NULL);
hButtonExit = CreateWindow("BUTTON","Exit",
    WS_CHILD | WS_VISIBLE |
    BS_PUSHBUTTON,
    0,0,140,20,hWindow,(HMENU) 2,
    hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);
SetTimer (hWindow, 1, 10, NULL);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
    WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_COMMAND:

```

```

        DestroyWindow (hWnd);
        return 0;
    case WM_TIMER:
        if (XPos < 200)
        {
            XPos++;
        }
        else
        {
            XPos = 0;
        }
        if (YPos < 40)
        {
            YPos = YPos + 4;
        }
        else
        {
            YPos = 0;
        }
        SetWindowPos (hWnd, HWND_TOP, XPos, YPos,
                     400-(XPos*2), 400-(YPos*2),
                     SWP_SHOWWINDOW);

        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc (hWnd, uiMessage,
                              wParam, lParam);
}
}

```

### 12.2.2 Erläuterung

#### SetTimer

Die Funktion `SetTimer` wird hier dazu benutzt, in bestimmten Zeitabständen `WM_TIMER` an ein Fenster zu senden.

```

UINT SetTimer( HWND hWnd, UINT nIDEvent,
              UINT uElapse, TIMERPROC lpTimerFunc);

```

- `hWnd` ist der Handle auf das Fenster, welches die Nachricht `WM_TIMER` empfangen soll.
- `nIDEvent` ist ein Wert, der dazu benutzt wird, Timer-Nachrichten zu identifizieren, wenn mehrere Timer vorhanden sind.
- `uElapse` ist die Zeit in Millisekunden, in der die `WM_TIMER`-Nachrichten gesendet werden.
- `lpTimerFunc` wird auf `NULL` gesetzt, um nur an ein Fenster Nachrichten zu senden.

## 12.3 Eine Timer-Anwendung mit einer Timer-Funktion

### 12.3.1 Quelltext

Diese Anwendung bewirkt dasselbe, wie die erste Timer-Anwendung. Sie sendet aber keine Timer-Nachrichten, sondern führt in bestimmten Zeitabständen eine Timer-Funktion aus.

```
#include <windows.h>

int XPos, YPos;
HWND hWnd;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
VOID CALLBACK TimerProc(HWND, UINT, UINT, DWORD);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    XPos = 0;
    YPos = 0;

    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfnWndProc = WndProc;
    WndClass.hInstance = hInstance;
```

```

WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hButtonExit;
hWindow = CreateWindow("WinProg","Fenster",
                      WS_OVERLAPPEDWINDOW,
                      XPos,YPos,
                      400-XPos,400-YPos,NULL,NULL,
                      hInstance, NULL);
hButtonExit = CreateWindow("BUTTON","Exit",
                           WS_CHILD | WS_VISIBLE |
BS_PUSHBUTTON,
                           0,0,140,20,hWindow,(HMENU) 2,
                           hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);

SetTimer (NULL, NULL, 10, TimerProc);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{

```



```
switch(uiMessage)
{
    case WM_COMMAND:
        DestroyWindow (hWnd);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc (hWnd, uiMessage,
                               wParam, lParam);
}

VOID CALLBACK TimerProc(HWND hWnd, UINT uMsg,
                       UINT idEvent, DWORD dwTime)
{
    if (XPos < 200)
    {
        XPos++;
    }
    else
    {
        XPos = 0;
    }
    if (YPos < 40)
    {
        YPos = YPos + 4;
    }
    else
    {
        YPos = 0;
    }
    SetWindowPos (hWindow, HWND_TOP, XPos,
                  YPos, 400-(XPos*2), 400-(YPos*2),
                  SWP_SHOWWINDOW);
}
```

### 12.3.2 Erläuterung

Die Funktion `SetTimer` wird verwendet, damit eine Funktion in bestimmten Zeitabständen aufgerufen wird.

```
UINT SetTimer( HWND hWnd, UINT nIDEvent,  
              UINT uElapse, TIMERPROC lpTimerFunc);
```

- `hWnd` wird auf `NULL` gesetzt.
- `nIDEvent` wird auf `NULL` gesetzt.
- `uElapse` bestimmt den Zeitabstand, in der die Timer-Funktion aufgerufen wird.
- `lpTimerFunc` ist ein Zeiger auf die Timer-Funktion.

# 13

## Der Drucker

---

### 13.1 Allgemeines

Der Drucker wird unter Windows über einen Gerätekontext-Handle angesprochen. Und auf den Drucker wird mit den gleichen Grafikbefehlen der GDI gezeichnet wie auf den Monitor. Dabei kommt es zu einer ausgiebigen Kommunikation mit dem Druckertreiber, der später für die Übermittlung der Daten an den Drucker zuständig ist. Zudem gibt es auch noch weitere Funktionen, die den Druck- und den Seitenanfang signalisieren.

### 13.2 Ein Druckerprogramm

#### 13.2.1 Der Quelltext

Dieses Programm ist in der Lage, bei einem Klick auf die Schaltfläche „Drucken“ mit einem Standarddruckerdialog eine Abfrage des Druckers vorzunehmen. Dann wird das Wort „Drucker“ auf dem Blatt ausgegeben. Der Drucker sollte zuvor auf „Grafikausgabe“ gestellt werden.

---



Bild 13.1: Die Anwendung nach dem Start

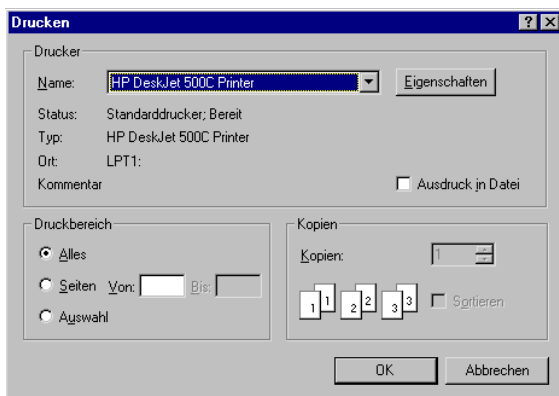


Bild 13.2: Der Standarddruckerdialog

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

```
int APIENTRY WinMain(HINSTANCE hInstance,
```

```

        HINSTANCE hPrevInstance,
        LPSTR      lpCmdLine,
        int        nCmdShow )
{

    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);
    HWND hWindow,hButton;
    hWindow = CreateWindow("WinProg","Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0,0,400,400,NULL,NULL,
                          hInstance, NULL);
    hButton = CreateWindow("BUTTON","Drucken",
                          WS_CHILD | WS_VISIBLE
                          | BS_PUSHBUTTON,
                          10,340,140,20,hWindow,(HMENU) 1,
                          hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    MSG Message;

    while (GetMessage(&Message, NULL, 0, 0))
    {
        DispatchMessage(&Message);
    }
}

```

```

    }

    return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam,LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_COMMAND:
            if (HIWORD(wParam) == BN_CLICKED)
            {
                if (LOWORD(wParam) == 1)
                {
                    PRINTDLG pd;
                    ZeroMemory(&pd, sizeof(PRINTDLG));
                    pd.lStructSize = sizeof(PRINTDLG);
                    pd.hwndOwner   = hWnd;
                    pd.hDevMode    = NULL;
                    pd.hDevNames   = NULL;
                    pd.Flags       = PD_USEDEVMODECOPIESANDCOLLATE
                                    | PD_RETURNDC;
                    pd.nCopies     = 1;
                    pd.nFromPage   = 0xFFFF;
                    pd.nToPage     = 0xFFFF;
                    pd.nMinPage    = 1;
                    pd.nMaxPage    = 0xFFFF;

                    if (PrintDlg(&pd) == true)
                    {
                        MessageBox (hWnd, "richtig", "richtig",MB_OK);
                    }
                }
            }
        }
    }
}

```

```
DOCINFO di;
di.cbSize = sizeof(DOCINFO);
di.lpszDocName = "Bitmap Printing Test";
di.lpszOutput = (LPTSTR) NULL;
di.fwType = 0;
StartDoc(pd.hDC, &di);
StartPage(pd.hDC);

TextOut(pd.hDC, 10, 10, "Drucker",
        lstrlen ("Drucker"));

EndPage(pd.hDC);

EndDoc(pd.hDC);

DeleteDC(pd.hDC);

}
}
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}
```

### 13.2.2 Erläuterung

Die einzigen Neuerungen liegen in der Nachricht WM\_COMMAND.

#### Aufruf des Druckerstandarddialogs

Windows stellt mehrere Standarddialoge zur Verfügung: für „Datei öffnen“, „Datei speichern“, „Farbauswahl“ und „Drucken“. In diesem Programm wird der Standarddialog für Drucken verwendet. Dieser Dialog ermöglicht die Auswahl des Druckers und spezifische Einstellungen. Der Dialog erstellt auch ein Gerätekontext-Objekt. Den Druckerstandarddialog erzeugt man mit der Funktion PrintDlg. Diese Funktion erwartet eine Struktur vom Typ PRINTDLG.

```
typedef struct tagPD
{
    pd DWORD lStructSize;
    HWND hwndOwner;
    HANDLE hDevMode;
    HANDLE hDevNames;
    HDC hDC;
    DWORD Flags;
    WORD nFromPage;
    WORD nToPage;
    WORD nMinPage;
    WORD nMaxPage;
    WORD nCopies;
    HINSTANCE hInstance;
    DWORD lCustData;
    LPPRINTHOOKPROC lpfnPrintHook;
    LPSETUPHOOKPROC lpfnSetupHook;
    LPCTSTR lpPrintTemplateName;
    LPCTSTR lpSetupTemplateName;
    HANDLE hPrintTemplate;
    HANDLE hSetupTemplate;
} PRINTDLG;
```

Wir gehen hier nur auf die Variablen ein, die auch verwendet werden. Alle anderen werden durch `ZeroMemory` auf `NULL` gesetzt.



- `lStructSize` ist die Größe der Struktur. Sie wird mit `sizeof` ermittelt.
- `HDC` ist ein Handle auf ein Gerätekontext-Objekt, das vom Dialog `PrintDlg` erzeugt wurde.
- `Flags` sind eine Reihe von Konstanten, die hier angegeben werden. `PD_RETURNDC` ist ein Flag, das bestimmt, dass ein Handle auf das Gerätekontext-Objekt für den Drucker zurückgegeben wird.
- `NCopies` ist die Anzahl der Kopien, die gedruckt werden sollen.

Nun muss noch mit dem Befehl `PrintDlg` ein Standarddialogfeld aufgerufen werden. Windows stellt einige Standarddialogfelder zur Verfügung. Dialogfelder sind nichts anderes als Fenster, die zur Eingabe bestimmt sind. Diese Standarddialogfelder werden mit einer Struktur aufgerufen und liefern Werte zurück. In diesem Programm wird der Druckerdialog verwendet. Er wird mit dem Befehl `PrintDlg` aufgerufen, dem eine Struktur übergeben wird, die zur Initialisierung und Ausgabe genutzt wird.

```
BOOL PrintDlg( LPPRINTDLG lppd);
```

Diesem Aufruf wird im Quelltext die oben besprochene Struktur übergeben.

```
PrintDlg(&pd);
```

## Einen Druckauftrag einleiten

Windows stellt Funktionen zur Verfügung, mit denen auf dem Drucker zugegriffen werden kann. Zum einen gibt es die Funktion, dass man den Drucker genau so behandeln kann wie eine Grafikausgabe, indem man einfach einen Gerätekontext-Handle erstellt. Mit den anderen Funktionen wird der Druckauftrag eingeleitet, der dann im Druckermanager erscheint. Dazu wird die Funktion `StartDoc` verwendet. Dann muss noch mit der Funktion `StartPage` eine neue Seite eingerichtet werden. Schon seit der Verwendung der Funktion `PrintDlg` ist der Gerätekontext mit dem Druckertreiber verbunden. Der Druck beginnt aber erst, nachdem der gesamte Druckauftrag abgeschlossen ist. Nachdem nun der Gerätekontext erstellt worden ist, werden die einzelnen Funktionen für den Drucker aufgerufen.

```
DOCINFO di;
    di.cbSize = sizeof(DOCINFO);
    di.lpszDocName = "Bitmap Printing Test";
    di.lpszOutput = (LPTSTR) NULL;
    di.fwType = 0;
    StartDoc(pd.hDC, &di);
    StartPage(pd.hDC);
```

```
TextOut(pd.hDC, 10, 10, "Drucker",
        lstrlen ("Drucker"));
```

```
EndPage(pd.hDC);
```

```
EndDoc(pd.hDC);
```

```
DeleteDC(pd.hDC);
```

Die Funktion `StartDoc` startet einen neuen Druckauftrag.

```
int StartDoc( HDC hdc, CONST DOCINFO *lpdi);
```

- `Hdc` ist der Handle auf das Geräte Kontext Objekt, für das ein neuer Druckjob angelegt werden soll.
- `Lpdi` ist ein Zeiger auf eine Struktur vom Typ `DOCINFO`. Diese Struktur hat für dieses Programm keine Bedeutung. Sie beinhaltet lediglich Angaben über Namen von Dokumenten und Dateien. Es muss nur die Größe angegeben werden. Die erforderliche Strukturvariable heißt `cbSize`.

Die Funktion `StartPage` beginnt eine neue Seite, in die dann mit den gewohnten GDI-Funktionen gezeichnet werden kann.

```
int StartPage( HDC hdc);
```

- `Hdc` ist der Handle auf das Gerätekontext-Objekt, das die neue Seite bekommt.

Die Funktion `EndPage` schließt eine Seite ab.

```
int EndPage( HDC hdc);
```

- `Hdc` ist der Handle auf das Geräte Kontext Objekt.

Die Funktion `EndDoc` beendet den gesamten Druckauftrag.

```
int EndDoc( HDC hdc);
```

- `Hdc` ist der Handle auf das Gerätekontext-Objekt.
- Nachdem mit `StartDoc` und `StartPage` eine Seite für einen Druckauftrag erstellt wurde, kann mit den GDI-Funktionen auf die Seite gezeichnet werden. Bevor man aber die GDI-Funktionen anwendet, muss man normalerweise festlegen, mit welcher Maßeinheit der Gerätekontext angesprochen werden soll.



TEIL II

Nitty  
Nitty  
Gritty

TAKE THAT!



# 14 Win32-API: Datentypen

## 14.1 Allgemeines

Die Datentypen, die in der Win32-API enthalten sind, definieren nicht nur die Größe und Art, sondern auch die Verwendung des Datentyps. So kommt es, dass viele Datentypen die gleiche Größe haben und numerisch sind. Dieses Kapitel bietet eine Auflistung wichtiger Datentypen, die in der Win32-API definiert sind, und in den Befehlen in diesem Buch verwendet werden.

## 14.2 Tabelle

Datentyp	Art	Verwendung
BOOL	boolean	Rückgabewerte
BYTE	8-Bit unsigned Integer	Standard 8 Bit unsigned Integer Variable.
COLORREF	32-Bit unsigned Integer	Dieser Datentyp wird für RGB-Farbwerte benutzt. Sie werden mit 0x00bbggrr im Wert angegeben.
DWORD	32-Bit unsigned Integer	Standard 32-Bit unsigned Integer Variable.
HANDLE	32-Bit	Allgemeiner Handle auf ein Objekt. Segment, das im Speicher aktiviert wird.
HBITMAP	32-Bit	Handle auf ein Bitmap-Objekt.
HBRUSH	32-Bit	Handle auf ein Brush-Objekt.
HCURSOR	32-Bit	Handle auf ein Cursorobjekt.
HDC	32-Bit	Handle auf einen Gerätekontext, also ein Device-Context-Objekt.

Datentyp	Art	Verwendung
HFILE	32-Bit	HANDLE auf ein Dateiobjekt, also ein File-Objekt.
HFONT	32-Bit	Handle auf ein Font-Objekt.
HGDIOBJ	32-Bit	Spezialisierter, allgemeiner Handle für GDI-Objekte.
HICON	32-Bit	Handle auf ein Icon-Objekt.
HINSTANCE	32-Bit	Handle auf eine Instanz.
HMENU	32-Bit	Handle auf ein Menüobjekt.
HPEN	32-Bit	Handle auf ein Pen-Objekt.
HWND	32-Bit	Handle auf ein Fensterobjekt.
LONG	32-Bit signed Integer	Standard 32-Bit signed Integer Variable.
LPARAM	32-Bit	Dies ist ein Parameter, der bei der Nachrichtenübergabe genutzt wird.
LPCSTR	Zeiger des Typs const char	Wird für Texte genutzt.
LPSTR	Zeiger des Typs char	Wird für Texte genutzt.
PVOID	Zeiger auf einen beliebigen Variablentyp	Standard für einen allgemeiner Zeiger.
TIMERPROC	Zeiger auf eine Funktion	Zeiger auf eine Funktion für den Timer.
UINT	32-Bit unsigned	Allgemein für Ganzzahlen.
WNDPROC	Zeiger auf eine Funktion	Zeiger auf eine Funktion, die die Nachrichtenverarbeitung übernimmt.
WORD	16-Bit unsigned Integer	Standard 16-Bit unsigned Integer Variable
WPARAM	32-Bit	Dies ist ein Parameter, der bei der Nachrichtenübergabe genutzt wird.



# 15 Funktionen, Strukturen, Nachrichten und Objekte der Win32-API

## 15.1 Allgemeines

Die folgenden Kapitel beschreiben die einzelnen Funktionen, die in diesem Buch bisher verwendet wurden und Funktionen, die noch nicht genutzt wurden, aber sehr wichtig sind. Mit diesen Funktionen werden jeweils auch die entsprechenden Strukturen, Nachrichten und Objekte beschrieben. Alles ist danach geordnet, wozu die Funktionen eingesetzt werden.

Bei jedem Funktionsparameter und Rückgabewert stehen die Buchstaben E, A, R und N. Sie geben jeweils die häufigste Verwendung an. Zeiger auf Daten können als Eingabe und Ausgabe definiert werden.

- E bedeutet, dieser Wert wird von der Funktion genutzt. Der Wert dient der Funktion als *Eingabe*.
- A bedeutet, dieser Wert wird von der Funktion belegt. Der Wert dient also als *Ausgabe* der Funktion.
- R heißt, dass der Wert ist keine Ausgabe über die Parameterliste der Funktion ist, sondern ein *return*-Wert.
- N ist ein Unterpunkt. Dieser Wert kann NULL sein. Die Funktion, die dieser Wert dann hat, wird entsprechend dargestellt.

Außerdem sind zu jeder Funktion, Struktur und Nachricht die Header-Datei und falls notwendig die Library-Datei sowie die Unterstützung der verschiedenen 32-Bit-Betriebssysteme und von ANSI- bzw. UNICODE angegeben. Falls dies nicht erwähnt wird, wird nur der ANSI-Code unterstützt - oder eine Unterstützung ist nicht notwendig, da in der Funktion keine Textzeichen verwendet werden.

Zum Schluss gibt es zu verschiedenen Funktionen oder Strukturen eine Liste der unterstützten Systeme.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Diese Liste zeigt an, unter welchen Betriebssystemen die Funktionen laufen. Programme, die unter Windows 95 laufen, funktionieren, ohne dass sie neu kompiliert werden, auch unter Windows NT, wenn alle Funktionen unterstützt werden. Werden die Funktionen alle auch von Windows CE unterstützt, laufen die Programme dort aber nur, wenn alles neu kompiliert wird, da sich die Prozessoren unterscheiden. Das Vorhandensein eines anderen Prozessors bedeutet nicht, dass sich das Prinzip des Windowsbetriebssystems ändert; es ändert sich nur der Code der einzelnen Funktionen, die aber dennoch alle vorhanden sind.

Es gibt noch weitere Zusatzinformationen:

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

Unterstützung von ANSI/UNICODE

In der Header-Datei sind die Informationen der Funktion oder Struktur gespeichert. Diese muss meistens nicht extra eingebunden werden, da sie normalerweise schon in die Datei `WINDOWS.H` eingebunden ist. Die Library-Datei ist die Datei mit dem bereits kompilierten Code. Diese Datei muss dem Compiler nicht mitgeteilt werden, da vor allem die Windowsgrundlagen benötigt werden. Diese werden automatisch in Windowsprogramme eingebunden. ANSI/UNICODE bestimmt, wie der Text, der in die Funktion eingegeben werden soll, gespeichert sein muss. Beides kann unterstützt werden und der Compiler wählt die richtige Funktion aus. Falls nur eine Variante unterstützt wird, muss manchmal durch das Makro `TEXT ANSI` in `UNICODE` umgewandelt werden.

# 16 Win32-API: Windows-Grundlagen

## 16.1 Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die Windows-Grundlagen

Das grundlegende Windowskonzept besteht aus Fenstern. Mit ihrer Hilfe kann man mehrere laufende Programme grafisch ausgeben. Zu diesem Zweck wurde eine Fensterhierarchie eingeführt. Selbst das Desktop-Window ist nur ein Fenster. Weiterhin wurde ein Nachrichtenkonzept entwickelt, mit dem Fenster untereinander kommunizieren können. Für die grafische Ausgabe benötigt man die GDI-Funktionen.

<b>Funktionen, Strukturen, Nachrichten und Objekte</b>	<b>Bedeutung</b>
Fenster-Objekt	Der wichtigste Punkt der Fenstertechnik.
CreateWindow	Erstellt ein neues Fenster.
WM_CREATE	Nachricht die nach der Erstellung des Fenster-Objekts gesendet wird.
CREATESTRUCT	Struktur für die Nachricht WM_CREATE.
DestroyWindow	Löscht ein Fenster-Objekt.
WM_DESTROY	Nachricht die nach der Zerstörung des Fenster-Objekts gesendet wird.
ShowWindow	Setzt den Anzeigestatus.
GetMessage	Holt Nachrichten aus der Nachrichtenwarteschlange und wartet auf weitere Nachrichten.
PeekMessage	Holt Nachrichten aus der Nachrichtenwarteschlange und wartet nicht auf weitere Nachrichten.

Funktionen, Strukturen, Nachrichten und Objekte	Bedeutung
RegisterClass	Registriert eine Fensterklasse.
WNDCLASS	Struktur, die von der Funktion RegisterClass zur Registrierung der Fensterklasse genutzt wird.
SetWindowText	Setzt den Text eines Fenster-Objekts.
TranslateMessage	Übersetzt mit dem Tastaturtreiber Virtual-Key-Nachrichten in Character-Nachrichten.
DispatchMessage	Ruft die Fensterfunktion auf und übergibt die Nachricht.
WM_PAINT	Wird gesendet, wenn ein Bereich des Fensters neu gezeichnet werden soll.

Tabelle 16.1: Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die Windows-Grundlagen

### 16.1.1 Fensterobjekt

Unter Windows dreht sich alles um das Fensterobjekt. Es ist, wie alle anderen Objekte auch, eine Datenstruktur, für die es bestimmte Verwaltungsfunktionen gibt. Dieses Objekt enthält unter anderem Daten über die Position, die Größe, die Eigenschaften oder die Stellung in der Hierarchie und kommuniziert nur mit Fensterobjekten. Diese Kommunikation betreiben das System und andere Programme, die ebenfalls Fensterobjekte besitzen. Fensterobjekte sind unter Windows der einzige Weg, eine Grafikausgabe mehrerer Anwendungen gleichzeitig zu realisieren. Fensterobjekte sind auch der einzige Weg, vom System, also auch von der Peripherie, Nachrichten zu bekommen.

Ein Fensterobjekt wird im Allgemeinen durch die Funktion `DefWindowProc`, als Fenster auf dem Bildschirm einheitlich dargestellt.

### 16.1.2 CreateWindow

Die Funktion `CreateWindow` erstellt ein Fensterobjekt.

```
HWND CreateWindow(LPCTSTR lpClassName,
                  LPCTSTR lpWindowName,
```

```

DWORD dwStyle,
int x,
int y,
int nWidth,
int nHeight,
HWND hWndParent,
HMENU hMenu,
HANDLE hInstance,
LPVOID lpParam
);

```

- **E: lpClassName** bestimmt den Namen der Fensterklasse.
- **E: lpWindowName** legt den Fenstertext fest.
- **E: dwStyle** bestimmt die Eigenschaften des Fensters. Hierzu können Konstanten über Oder-Verbindungen angegeben werden. Die Eigenschaften haben mehrere Aufgaben. Zum einen legen sie fest, wie der Größenunterschied des Fenster-Gerätekontexts zum Client-Gerätekontext ist. Der Fenster-Gerätekontext umfasst alles: das gesamte Fenster mit Titelleisten, Menüs, usw. Außerdem wird über Fensterregionen festgelegt, welche Teile im entsprechenden Gerätekontext überhaupt verändert werden dürfen. Regionen sind für einen Gerätekontext die Flächen, in die gezeichnet werden darf. Es können zusätzlich benutzerdefinierte Regionen zugewiesen werden. Diese sind den Fensterregionen aber untergeordnet. Fensterregionen werden danach definiert, welcher Bereich z. B. neu gezeichnet werden muss. Auch das Verhalten von Fenstern, wann sie Nachrichten geben und wann überhaupt etwas neu gezeichnet wird, wird durch diese Einstellungen festgelegt.

WS\_BORDER bestimmt, dass das Fenster einen Rahmen hat.

WS\_CAPTION definiert, dass das Fenster eine Titelleiste hat. Diese Konstante beinhaltet WS\_BORDER.

WS\_CHILD/WS\_CHILDWINDOW legt fest, dass das Fenster ein Fenster unterer Ordnung ist.

`WS_CLIPCHILDREN` bestimmt, dass die Bereiche von Fenstern unterer Ordnung beim Zeichnen nicht überschrieben werden können.

`WS_CLIPSIBLINGS` bestimmt, dass die Bereiche von Nachbarfenstern unterer Ordnung nicht überzeichnet werden können.

`WS_DISABLED` definiert, dass das Fenster keinen Input mehr vom System bekommt.

`WS_DLGFRAAME` legt fest, dass das Fenster einen Rand wie ein Dialogfeld hat.

`WS_GROUP` bewirkt, dass das Fenster zu einer Gruppe gehört.

`WS_HSCROLL` bestimmt, dass das Fenster einen horizontalen Scrollbalken hat.

`WS_ICON1/WS_MINIMIZE` erreicht, dass das Fenster minimiert initialisiert wird.

`WS_MAXIMIZE` bestimmt, dass das Fenster maximiert initialisiert wird.

`WS_MAXIMIZEBOX` bedeutet, dass das Fenster mit einem Maximieren-Button dargestellt wird. `WM_SYSMENU` muss auch aktiviert sein.

`WS_MINIMIZEBOX` legt fest, dass das Fenster mit einem Minimieren-Button dargestellt wird. `WM_SYSMENU` muss auch aktiviert sein.

`WS_OVERLAPPED/WS_TILED` setzt fest, dass das Fenster ein überlappendes Fenster ist. Ein überlappendes Fenster ist ein Fenster, das von anderen verdeckt werden und selber andere Fenster in seinem Bereich verdecken kann. Dieses Fenster hat eine Titelleiste und einen Rahmen.

`WS_OVERLAPPEDWINDOW/WS_TILEDWINDOW` bestimmt, dass das Fenster sich aus mehreren Konstanten zusammensetzt. Diese Konstanten sind:



`WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`, `WS_THICKFRAME`, `WS_MINIMIZEBOX` und `WS_MAXIMIZEBOX`.

WS\_POPUP definiert, dass das Fenster ein Popup-Fenster ist. Ein Popup-Fenster verhält sich wie ein Overlapped-Fenster. Es hat aber keinen Rahmen und keine Titelleiste, sondern stellt nur die Fläche dar.

WS\_POPUPWINDOW erklärt, dass das Fenster sich aus mehreren Konstanten zusammensetzt. Diese Konstanten sind: WS\_BORDER, WS\_POPUP und WS\_SYSMENU. Um ein Menü im Fenster zu haben, muss WS\_CAPTION aktiviert sein.

WS\_SIZEBOX/WS\_THICKFRAME bestimmt, dass das Fenster einen Size-Button hat.

WS\_SYSMENU bewirkt, dass das Fenster ein Fenstermenü in seiner Titelleiste hat. WS\_CAPTION muss dabei auch aktiviert sein.

WS\_TABSTOP bestimmt, dass das Fenster durch das Drücken der Taste  den Tastaturfokus erhalten kann. Wenn die Taste  gedrückt wird, bekommt das nächste Fenster auf der gleichen Ebene mit WS\_TABSTOP den Tastaturfokus.

WS\_VISIBLE legt fest, dass das Fenster sichtbar initialisiert wird. Standardmäßig werden alle Fenster unsichtbar initialisiert.

WS\_VSCROLL bewirkt, dass das Fenster einen vertikalen Scrollbalken hat.

- E: X und Y bestimmen die obere linke Position des Fensters. Bei einem unabhängigen Fenster ist diese Position in Bildschirmkoordinaten anzugeben. Bei einem Child-Fenster ist die Position relativ zur oberen linken Ecke des Parent-Fensters. Alle Werte sind Pixelangaben.

Wenn bei Parameter X die Konstante CW\_USEDEFAULT angegeben wird, wird das Fenster auf die Standardposition gesetzt, und der Wert in Y wird ignoriert.

- E: nWidth und nHeight geben die Ausdehnung des Fensters in der Breite und Höhe in Pixeln an.

Wenn bei Parameter nWidth die Konstante CW\_USEDEFAULT angegeben wird, wird das Fenster von seinen Ausdehnungen her auf

Standardwerte gesetzt. Dabei wird dann der Wert in `nHeight` ignoriert.

- E: `hWndParent` gibt den Handle für ein Parent-Fenster an. Ein Parent-Fenster ist das Fenster, welches einem anderen Fenster übergeordnet ist. Diese Überordnung bezieht sich auf die grafische Anzeige und auf die Interaktion. Das Gegenstück zum Parent-Fenster ist das Child-Fenster.
- N: Wenn dieser Wert NULL ist, ist das Fenster unabhängig und besitzt kein übergeordnetes Fenster.
- E: `hMenu` ist ein Handle auf ein Menü-Objekt. Dieses Menü-Objekt wird dazu genutzt, ein Menü im Fenster anzuzeigen. Dieser Wert wird sehr oft verwendet, um für Fenster von Steuerelementen eine Identifikationsnummer anzugeben. Dazu muss dieser Wert erst in einen Handle auf ein Menü-Objekt umgewandelt werden.
- N: Wenn dieser Wert NULL ist, hat das entsprechende Fenster kein Menü.
- E: `hInstance` ist ein Handle auf die Instanz, zu der das Fenster gehört. Dieser Parameter drückt nur die Zugehörigkeit des Fensters zu einem Thread aus. Er hat keine Auswirkungen auf das Objekt an sich. Man verwendet ihn z.B. um anzugeben, in welcher Nachrichtenwarteschlange welches Thread die Nachrichten von Benutzereingaben für ein Fenster eingetragen werden sollen.
- E: `lpParam` gibt einen Wert an, der der Struktur `CREATESTRUCT` hinzugefügt wird. Diese Struktur wird als Nachricht `WM_CREATE` für das Fensterobjekt in die Nachrichtenwarteschlange des Threads eingefügt.
- N: Dieser Parameter kann NULL sein. Dann wird einfach der Wert NULL als Parameter für die `WM_CREATE` Nachricht übergeben.
- R/A: Wenn die Funktion ein Fenster erstellt hat, liefert sie als `return` den Handle auf das erstellte Fensterobjekt zurück.

Wenn kein Fensterobjekt erstellt wurde, liefert die Funktion NULL zurück.



Die genaue Bedeutung der Fehlermeldung erfährt man über die Funktion `GetLastError`.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

ANSI/UNICODE: Unterstützung für beides unter Windows NT

### 16.1.3 WM\_CREATE

Die Nachricht `WM_CREATE` wird in eine Nachrichtenwarteschlange eingefügt, wenn für die Nachrichtenwarteschlange dieses Threads ein Fenster erstellt wurde. Dabei wird die Nachricht natürlich an das Fenster gesendet.

`WM_CREATE`

`lpcs = (LPCREATESTRUCT) lParam;`

- **A:** `lpcs` enthält einen Zeiger auf eine Struktur vom Typ `CREATESTRUCT`. Diese Struktur enthält Angaben über das erstellte Fenster.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

### 16.1.4 CREATESTRUCT

Diese Struktur wird der Nachricht `WM_CREATE` übergeben. Eine Variable dieser Struktur wird von der Funktion `CreateWindow` gesetzt.

```
typedef struct tagCREATESTRUCT
```

```
{  
    LPVOID lpCreateParams;
```

```

HINSTANCE hInstance;
HMENU hMenu;
HWND hwndParent;
int cy;
int cx;
int y;
int x;
LONG style;
LPCTSTR lpszName;
LPCTSTR lpszClass;
DWORD dwExStyle;
} CREATESTRUCT;

```

- A: `lpCreateParams` enthält den Parameter, der in der Funktion `CreateWindow` angegeben wurde.
- A: `hInstance` enthält den Handle auf den Thread, zu dem das Fenster gehört.
- A: `hMenu` enthält den Handle auf das Menü-Objekt, das für das Fenster angegeben wurde.
- A: `hwndParent` enthält den Handle auf das Fensterobjekt, das als höher gestelltes Fenster für das Fenster angegeben wurde.
- A: `cy`, `cx`, `y` und `x` enthalten die Werte über Position und Ausdehnung des Fensters. Dabei können diese Werte relativ oder absolut auf dem Bildschirm erscheinen.
- A: `style` enthält die Eigenschaften des Fensters.
- A: `lpszName` enthält den Namen des Fensters.
- A: `lpszClass` enthält den Namen der Fensterklasse, auf der das Fenster basiert.
- A: `dwExStyle` gibt die Eigenschaften des Fensters wieder, die von der Funktion `CreateWindowEx` zusätzlich festgelegt werden. Diese sind die `WS_EX` Styles.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

immer als ANSI/UNICODE verfügbar

### 16.1.5 DestroyWindow

Die Funktion `DestroyWindow` löscht ein Fensterobjekt. Zuvor werden die Nachrichten `WM_DESTROY` und `WM_NCDESTROY` in die Warteschlange des Threads eingefügt, zu dem das Fenster gehört. Dies geschieht, damit das Fenster deaktiviert werden und seinen Eingabefokus verlieren kann. Die Funktion löscht auch das Menü des Fensters, räumt die Warteschlange des Threads auf, löscht den Timer und gibt das Clipboard wieder frei.

```
BOOL DestroyWindow( HWND hWnd );
```

- E: `hWnd` ist der Handle auf das Fensterobjekt, welches gelöscht werden soll.
- A/R: Wenn die Funktion das Fensterobjekt gelöscht hat, ist der `return` ungleich `NULL`.

Wenn das Fensterobjekt nicht gelöscht wurde oder ein anderer Fehler aufgetreten ist, ist der Rückgabewert gleich `NULL`. Weitere Informationen über den Fehler bekommt man mit der Funktion `GetLastError`.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

### 16.1.6 WM\_DESTROY

Die Nachricht `WM_DESTROY` wird von der Funktion `DestroyWindow` an das Fenster gesendet, welches gelöscht werden soll. `WM_DESTROY` wird gesendet, bevor das Fensterobjekt gelöscht wird.

```
WM_DESTROY
```

Diese Nachricht besitzt keine Parameter.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

### 16.1.7 ShowWindow

Die Funktion `ShowWindow` bestimmt, wie das Fenster angezeigt wird. Für diesen Zweck wird das Fensterobjekt modifiziert. Die Werte dieser Funktion werden z.B. von der Funktion `GetDC` ausgewertet.

```
BOOL ShowWindow (HWND hWnd, int nCmdShow);
```

- E: `hWnd` ist der Handle auf das Fenster, von dem die Anzeigeeigenschaften geändert werden sollen.
- E: `nCmdShow` bestimmt, welcher Wert für den Anzeigestatus im Fensterobjekt gesetzt wird. Eine der folgenden Konstanten definiert, wie das Fenster angezeigt wird.

`SW_SHOW` aktiviert das Fenster und zeigt es mit den aktuellen Einstellungen an.

`SW_HIDE` versteckt das Fenster und aktiviert ein anderes.

`SW_MAXIMIZE` maximiert das Fenster.

`SW_MINIMIZE` minimiert das Fenster. Dabei wird dann das nächste Fenster in der Hierarchie aktiviert.

`SW_RESTORE` bewirkt fast dasselbe wie `SW_SHOW`. Diese Konstante sollte man benutzen, wenn ein Fenster aus dem minimierten Zustand wieder hergestellt werden soll.

`SW_SHOWDEFAULT` bestimmt, dass die Anzeige mit Daten gesetzt wird, die der Anwendung beim Start übergeben wurden. Eine Anwendung wird normalerweise mit der Funktion `CreateProcess` gestartet. Dort wird der Anzeigewert in der `STARTUP` Struktur übergeben.

SW\_SHOWMAXIMIZED legt fest, dass das Fenster aktiviert und maximiert wird.

SW\_SHOWMINIMIZED erreicht, dass das Fenster aktiviert und minimiert wird.

SW\_SHOWMINNOACTIVE bestimmt, dass das aktive Fenster aktiviert bleibt und dass es minimiert wird.

SW\_SHOWNA/SW\_SHOWNOACTIVE erklärt, dass das aktive Fenster aktiviert bleibt und dass es zur Anzeige die aktuellen Daten nutzt.

SW\_SHOWNORMAL aktiviert das Fenster und zeigt es mit den aktuellen Daten an.

- A/R: Wenn das Fenster vorher sichtbar war, liefert die Funktion ungleich NULL zurück.
- N: Wenn das Fenster vorher unsichtbar war, liefert die Funktion NULL zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

### 16.1.8 GetMessage

Die Funktion `GetMessage` holt die Nachrichten für ein bestimmtes Fenster und seine Child-Fenster aus der Nachrichtenwarteschlange. Dabei werden aber nur die Nachrichten für die Fenster des aufrufenden Threads bearbeitet. Nachrichten von anderen Fenstern können nicht bearbeitet werden. Es werden nur Nachrichten aus der Warteschlange des aufrufenden Threads geholt. Diese Funktion bildet mit anderen Funktionen normalerweise das Kernstück von Windowsprogrammen. Sie wird in Verbindung mit einer While-Schleife genutzt, um die Nachrichtenverarbeitung sicherzustellen. Durch die Nachricht `WM_QUIT` wird das Programm dann beendet. Falls keine Nachricht in

der Nachrichtenwarteschlange ist, sorgt die Funktion `GetMessage` dafür, dass das Programm so lange keine Prozessorzeit mehr bekommt, bis eine neue Nachricht in der Nachrichtenwarteschlange eintrifft. Dies ist der wesentliche Unterschied zur Funktion `PeekMessage`.

```
BOOL GetMessage( LPMSG lpMsg, HWND hWnd,
                UINT wMsgFilterMin, UINT wMsgFilterMax);
```

- A: `lpMsg` ist ein Zeiger auf eine MSG-Struktur. Diese Struktur enthält Informationen über die Art der Nachricht.
- E: `hWnd` ist der Handle auf ein Fenster von dem aufrufenden Thread, von dem die Nachrichten geholt werden sollen.
- N: Falls dieser Parameter NULL ist, werden alle Nachrichten für alle Fenster des aufrufenden Threads herausgesucht.
- E: `wMsgFilterMin` bestimmt den untersten Bereich der Nachrichten.
- N: Falls der angegebene Wert NULL ist, gibt es keine untere Grenze.
- E: `wMsgFilterMax` definiert den obersten Bereich der Nachrichten.
- N: Falls der angegebene Wert NULL ist, gibt es keine obere Grenze.
- A/R: Wenn die Funktion eine andere Nachricht als `WM_QUIT` heraussucht, liefert sie ungleich NULL zurück.

Falls die Funktion die Nachricht `WM_QUIT` heraussucht, liefert sie NULL zurück.

Falls ein Fehler bei der Ausführung der Funktion eintritt, liefert die Funktion `-1` zurück. Weitere Informationen über den Fehler bekommt man mit der Funktion `GetLastError`.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: USER32.LIB

immer als ANSI/UNICODE unter Windows NT verfügbar

### 16.1.9 PeekMessage

Die Funktion `PeekMessage` bearbeitet Nachrichten aus der Nachrichtenwarteschlange. Dabei kann die Funktion die Nachrichten aus der Warteschlange herausnehmen oder sie dort belassen. Sie sorgt nicht dafür, dass der Thread keine Prozessorzeit mehr bekommt, sondern der Thread wird wie gewöhnlich weiter ausgeführt. Darin liegt der Hauptunterschied zu `GetMessage`. Diese Funktion wird dann verwendet, wenn die Anwendung etwas bewirkt, dass durch bestimmte Ereignisse gesteuert werden soll. Eine solche Anwendung gibt es z.B. bei kleinen Spielen. Nachrichten werden nur für Fenster des aufrufenden Threads herausgesucht. Man kann angeben, für welches Fenster die Nachrichten bearbeitet werden sollen. Natürlich werden dann auch alle Nachrichten für die Child-Fenster herausgesucht.

```
BOOL PeekMessage(LPMSG lpMsg, HWND hWnd,  
                UINT wMsgFilterMin, UINT wMsgFilterMax,  
                UINT wRemoveMsg);
```

- A: `lpMsg` ist ein Zeiger auf eine MSG-Struktur. Diese Struktur enthält Daten über die Nachrichtenart.
- E: `hWnd` ist der Handle auf das Fensterobjekt, für das die Nachrichten herausgesucht werden sollen.
- N: Falls der Wert NULL ist, werden alle Nachrichten für den aufrufenden Thread aus der Nachrichtenwarteschlange herausgesucht.
- E: `wMsgFilterMin` bestimmt den unteren Wertebereich der Nachrichten.
- N: Falls dieser Wert NULL ist, gibt es keinen unteren Wertebereich.
- E: `wMsgFilterMax` bestimmt den oberen Wertebereich der Nachrichten.
- N: Falls dieser Wert NULL ist, gibt es keinen oberen Wertebereich.
- `wRemoveMsg` ist ein Wert, der durch Konstanten angegeben wird. Dieser Wert bestimmt, ob die Nachrichten aus der Warteschlange herausgeholt werden sollen, oder ob sie darin bleiben sollen. Eine

16

Nitty Gritty • Take that!

der folgenden Konstanten kann dafür alleine angegeben werden. Dazu kommen optional noch andere Konstanten, die angeben, ob eine bestimmte Art von Nachrichten aus der Nachrichtenwarteschlange geholt werden soll.

`PM_NOREMOVE` bestimmt, dass die Nachrichten in der Nachrichtenwarteschlange verbleiben.

`PM_REMOVE` setzt fest, dass die Nachrichten nicht in der Nachrichtenwarteschlange verbleiben.

Normalerweise werden alle Nachrichten aus der Nachrichtenwarteschlange herausgesucht. Diese Auswahl wird nur durch `hWnd` beeinflusst. Falls eine der folgenden Konstanten angegeben wird, wird die Auswahl beschleunigt. Eine oder mehrere dieser Konstanten können zusätzlich zu den oben genannten Konstanten angegeben werden.

Alle diese Konstanten können aber erst unter Windows 98 und Windows NT 5.0 deklariert werden.

`PM_QS_INPUT` bestimmt, dass aus der Nachrichtenwarteschlange Nachrichten, die durch einen Mausklick unter Nachrichten, die von der Tastatur ausgelöst wurden, herausgesucht werden sollen.

`PM_QS_PAINT` bewirkt, dass `PAINT`-Nachrichten herausgesucht werden sollen.

`PM_QS_POSTMESSAGE` bestimmt, dass alle Nachrichten, die mit `PostMessage` in die Warteschlange geschrieben worden sind herausgesucht werden. Hinzu kommt, dass alle Timer- und Hotkey-Nachrichten ausgelesen werden.

`PM_QS_SENDMESSAGE` bestimmt, dass alle Nachrichten, die mit der Funktion `SendMessage` in die Nachrichtenwarteschlange geschrieben worden sind, aus der Nachrichtenwarteschlange herausgesucht werden.

- A/R: Wenn eine Nachricht herausgenommen wurde, gibt die Funktion ungleich `NULL` zurück.



- **N:** Wenn keine Nachricht gefunden wurde, gibt die Funktion NULL als `return` zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

immer als ANSI/UNICODE unter Windows NT verfügbar

### 16.1.10 RegisterClass

Die Funktion `RegisterClass` registriert eine Fensterklasse. Diese Fensterklasse kann später von `CreateWindow` genutzt werden. Eine Fensterklasse ist eine Datenstruktur, die einige Eigenschaftsdaten setzt und das Fensterobjekt vorab mit einer Funktion zur Nachrichtenbehandlung belegt.

```
ATOM RegisterClass (CONST WNDCLASS *lpWndClass);
```

- **E:** `lpWndClass` ist ein Zeiger auf eine Struktur vom Typ `WNDCLASS`. Diese Struktur beinhaltet die Daten der Fensterklasse.
- **A/R:** Als `return` wird ein `ATOM` (ein Integer-Wert, der in einer Liste, einem `atom table`, auf einen String zeigt) zurückgeliefert. Der String identifiziert die Klasse.
- **N:** Falls ein Fehler auftritt, liefert die Funktion NULL zurück. Weitere Informationen über den Fehler bekommt man mit der Funktion `GetLastError`.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

immer als ANSI/UNICODE unter Windows NT verfügbar

### 16.1.11 WNDCLASS

Die Struktur WNDCLASS beinhaltet alle Daten, die einer Fensterklasse mitgegeben werden.

```
typedef struct _WNDCLASS
{
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HANDLE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

- **E:** `style` bestimmt die Klassen-, also Grundeigenschaften des Fensters. Dazu kommen die Eigenschaften von `CreateWindow`. Die Grundeigenschaften können aus einer Kombination der unten genannten Konstanten bestehen. Die Klasseneigenschaften sind aber nicht nur die Grundeigenschaften des Fensters, sondern sie bestimmen auch Eigenschaften der Klasse, z.B. ob die Klasse auch von anderen Threads zur Erstellung von Fenstern genutzt werden darf.

`CS_CLASSDC` gibt an, dass ein Gerätekontext von allen Fenstern in der Klasse genutzt wird.

`CS_DBLCLKS` bewirkt, dass eine Nachricht über einen Doppelklick an das Fenster gesendet wird, wenn der Cursor in diesem Fenster ist.

`CS_GLOBALCLASS` setzt fest, dass der Wert von `hInstance` für die Klasse unwichtig ist. Wenn diese Konstante nicht aktiviert ist, kann ein Fenster nur mit demselben Wert `hInstance` erstellt werden, den auch die Klasse hat.

16

Nitty Gritty • Take that!

CS\_HREADRAW bestimmt, dass das gesamte Fenster neu gezeichnet werden soll, wenn eine Positionsänderung oder eine Größenänderung in der Breite des Clientbereichs des Fensters durchgeführt wird.

CS\_VREDRAW definiert, dass das gesamte Fenster neu gezeichnet werden soll, wenn eine Positionsänderung oder eine Größenänderung in der Höhe des Clientbereichs des Fensters durchgeführt wird.

CS\_NOCLOSE bestimmt, dass „Schließen“ nicht mehr aus dem Fenstermenü ausgewählt werden kann.

CS\_OWNDCE erreicht, dass für jedes Fenster ein eigener Gerätekontext erstellt wird.

CS\_PARENTDC setzt fest, dass die Clipping-Region des Child-Fensters auf die Größe des Parent-Fensters gesetzt wird.

CS\_SAVEBITS legt fest, dass ein Fenster die Pixel, die es verdeckt, speichert. Diese Daten benutzt es, um die Grafik wiederherzustellen, falls es bewegt wird. Dies passiert aber nur dann, wenn in der Zwischenzeit keine Veränderungen an dem darunter liegenden grafischen Bereich vorgenommen worden sind. Dieses Verfahren hat den Vorteil, dass weniger WM\_PAINT-Nachrichten gesendet werden.

CS\_BYTEALIGNCLIENT bestimmt, dass der Clientbereich eines Fensters von der Position auf dem Bildschirm durch die Bytegrenze in horizontaler Richtung definiert wird.

CS\_BYTEALIGNWINDOW bewirkt, dass ein Fenster von der Position auf dem Bildschirm durch die Bytegrenze in horizontaler Richtung bestimmt wird.

- E: `lpfnWndProc` ist ein Zeiger auf eine Funktion. Diese Funktion ist die Fensterfunktion. Sie wird dafür verwendet, die Nachrichtenverarbeitung des Fensters zu übernehmen. Diese Funktion wird wiederum von der Funktion `DispatchMessage` aufgerufen.
- E: `cbClsExtra` legt die Menge an zusätzlichen Bytes fest, die der Klassenstruktur folgen, wenn sie registriert wird. Diese Bytes werden auf NULL gesetzt.

- E: `cbWndExtra` bestimmt die Menge an zusätzlichen Bytes, die dem erstellten Fensterobjekt folgen. Diese Bytes können mit `SetWindowLong` gesetzt werden, nachdem das Fenster erstellt worden ist.
- E: `hInstance` ist der Handle auf die Instanz, in der sich die Funktion für die Fensterklasse befindet.
- E: `hIcon` bestimmt den Handle auf ein Icon. Dabei ist dieses Icon eine Ressource. Ressourcen können auch während der Laufzeit aus Dateien erstellt werden. Sie müssen nicht unbedingt in der Anwendungsdatei vorhanden sein.
- N: Falls der Wert NULL ist, hat das Fenster ein Standardicon.
- E: `hCursor` bestimmt den Handle auf einen Cursor. Dabei ist dieser Cursor eine Ressource.
- N: Falls dieser Wert NULL ist, hat das Fenster keinen besonderen Cursor; der Cursor des Fensters ist dann der Cursor, der vorher gesetzt war.
- E: `hbrBackground` bestimmt den Handle auf ein Brush-Objekt. Dieses Brush-Objekt wird dazu verwendet, den Hintergrund des Fensters mit `BeginPaint` standardmäßig neu zu zeichnen. Es gibt eine Menge von vordefinierten System-Brush-Objekten. Diese Objekte definieren alle Farben. Es gibt aber nur zwanzig Systemfarben. Diese Systemfarben benennt man in der Systemsteuerung unter Darstellung.

COLOR\_ACTIVEBORDER  
COLOR\_ACTIVECAPTION  
COLOR\_APPWORKSPACE  
COLOR\_BACKGROUND  
COLOR\_BTNFACE  
COLOR\_BTNSHADOW  
COLOR\_BTNTEXT  
COLOR\_CAPTIONTEXT  
COLOR\_GRAYTEXT

COLOR\_HIGHLIGHT  
COLOR\_HIGHLIGHTTEXT  
COLOR\_INACTIVEBORDER  
COLOR\_INACTIVECAPTION  
COLOR\_MENU  
COLOR\_MENUTEXT  
COLOR\_SCROLLBAR  
COLOR\_WINDOW  
COLOR\_WINDOWFRAME  
COLOR\_WINDOWTEXT

- N: Falls dieser Wert auf NULL gesetzt wird, wird der Hintergrund des Fensters nicht neu gezeichnet
- E: `lpzMenuName` bestimmt den Ressourcennamen für ein Menü.
- N: Falls dieser Wert NULL ist, gibt es kein Menü für das Fenster. Jedes Fenster muss dann sein eigenes Menü zugewiesen bekommen.
- E: `lpzClassName` bestimmt den Namen der Fensterklasse. Dieser Parameter kann aber auch ein ATOM sein, das zurher mit `AddGlobalAtom` erstellt worden sein muss. Es handelt sich dabei nur um einen anderen Verweis auf den Namen.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

immer als ANSI/UNICODE verfügbar

### 16.1.12 TranslateMessage

Die Funktion `TranslateMessage` übersetzt die Nachrichten mit Virtual-key-Codes in Nachrichten mit Character-Codes. Mit dem Tasta-

turtreiber wird je nach Ländereinstellung aus den Scan-Codes ein Virtual-key-Code erzeugt. Dieser wird als Nachricht in die Nachrichtenwarteschlange eingefügt. Aus diesen Virtual-key-Codes werden je nach Ländereinstellungen durch die Funktion `TranslateMessage` ANSI-Codes erzeugt. Diese werden mit einer neuen Nachricht wieder in die Nachrichtenwarteschlange eingefügt.

```
BOOL TranslateMessage( CONST MSG *lpMsg);
```

- E: `lpMsg` ist ein Zeiger auf eine MSG-Struktur. Diese MSG-Struktur wird auf Nachrichten überprüft, die umgewandelt werden können. Deswegen steht diese Funktion auch meistens mitten in der While-Schleife, damit sie jede Nachricht überprüfen kann.
- A/R: Wenn die Nachricht übersetzt wurde, ist der Rückgabewert ungleich NULL.
- N: Wenn die Nachricht nicht übersetzt wurde, ist der Rückgabewert gleich NULL.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

### 16.1.13 DispatchMessage

Die Funktion `DispatchMessage` ruft die Fensterfunktion auf. Dieser Funktion übergibt sie dann auch die Nachricht.

```
LONG DispatchMessage( CONST MSG *lpmsg);
```

- E: `lpmsg` ist ein Zeiger auf eine MSG-Struktur. Diese MSG-Struktur enthält die Nachricht, die der Fensterfunktion übergeben wird, an die die Nachricht adressiert ist.
- A/R: Als `return` gibt die Funktion das wieder, was wiederum die Nachrichten als `return` zurückgeben.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

immer als ANSI/UNICODE unter Windows NT verfügbar

### 16.1.14 WM\_PAINT

Die Nachricht WM\_PAINT wird gesendet, wenn ein Bereich des Fensters neu gezeichnet werden soll. Der Nachricht wird der Gerätekontext des Clientbereichs des Fensters übergeben. Das Neuzeichnen von Rand und Titelleiste wird durch die Nachricht WM\_NCPAINT gesteuert. Dies wird von der Funktion `DefWindowProc` übernommen. Die Nachricht WM\_PAINT wird normalerweise selbst verarbeitet. Natürlich ist die Region des Gerätekontexts auf den Bereich eingegrenzt, der neu gezeichnet werden soll. Im Zusammenhang mit der Nachricht WM\_PAINT verwendet man die Funktionen `BeginPaint` und `EndPaint`, die die Bereiche, die neu gezeichnet werden sollen, wieder für gültig erklären.

WM\_PAINT

`hdc = (HDC) wParam;`

- `hdc` ist der Gerätekontext des Clientbereichs des Fensters, in dem ein bestimmter Bereich neu gezeichnet werden soll. Der Bereich wird angegeben durch eine Region, die mit höherer Priorität gesetzt wurde.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

## 16.2 Beispiele

### 16.2.1 Ein normales Fenster erstellen

Es wird ein normales Overlapped-Fenster erstellt.



Bild 16.1: Die Anwendung nach dem Start

#### Quelltext

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR     lpCmdLine,
                    int       nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
```

16

Nitty Gritty • Take that!



```

WndClass.cbWndExtra = 0;
WndClass.lpfWndProc = WndProc;
WndClass.hInstance = hInstance;
WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hWindow;
hWindow = CreateWindow("WinProg", "Fenster",
                      WS_OVERLAPPEDWINDOW,
                      0, 0, 600, 460, NULL, NULL,
                      hInstance, NULL);

ShowWindow (hWindow, nCmdShow);

UpdateWindow (hWindow);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_DESTROY:

```

```
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc (hWnd, uiMessage,
                               wParam, lParam);
    }
}
```

## 16

Nitty Gritty • Take that!

# 17

## Win32-API: GDI

### 17.1 Funktion, Strukturen, Nachrichten und Objekte der Win32-API für die GDI

Die GDI (*Graphics Device Interface*) ist eine Schnittstelle für grafische Geräte. Sie stellt Funktionen bereit, die aufgerufen werden können. Diese rufen dann Treiberfunktionen auf. Die Treiber wiederum müssen die Funktionen zur Verfügung stellen, die von der GDI aufgerufen werden.

Die GDI-Funktionen stellen eine Möglichkeit dar, mit der Grafikhardware des Computers zu kommunizieren ohne die Hardware direkt anzusprechen. Dies wird von der GDI in Zusammenarbeit mit den Treibern übernommen. Natürlich erfolgt die Grafikausgabe nicht nur auf dem Bildschirm über die Grafikkarte, sondern auch auf dem Drucker. Dies sind die Hauptgeräte der GDI-Funktionen. Beide Geräte können aber nur geräte-unabhängig angesprochen werden, weil die Treiber die Geräte identisch aussehen lassen.

Über GDI-Funktionen bekommt man Gerätekontexte oder besser Gerätekontext-Objekte. Diese Gerätekontext-Objekte können sich nur auf Teilbereiche des Gerätes und seiner Ausgabe beziehen. Nur in diesen Gerätekontexten ist es möglich, Grafikfunktionen anzuwenden. Das Wort Gerätekontext steht hier für ein Gerätekontext-Objekt, das den Bereich eines Gerätes und dessen Ausgabe beschreibt. Dabei können Gerätekontexte zwei Arten von Regionen (engl. *Regions*) beinhalten. Eine Region ist der Bereich, in dem die Grafikfunktionen sichtbar werden. Die GDI ist also die Verbindung zur Grafikhardware und ermöglicht die Ausgabe von Grafik. Sie ist ein Grundbestandteil der Win32-API.

Die GDI ist notwendig, um die grafische Ausgabe des Fensterkonzeptes zu realisieren. Man kann von jedem Fenster einen Gerätekontext ermitteln. Die Regionen des Gerätekontextes sind so gesetzt, dass nur in die Bereiche gezeichnet wird, die neu erstellt werden müssen. Dass ein Bereich eines Fensters neu gezeichnet werden muss, erfährt ein Fenster durch ein anderes. Dieser Fall kann zum Beispiel eintreten, wenn sich die Fenster bewegen und ein Fenster ein Stück eines anderen frei werden lässt.

Funktionen, Strukturen, Nachrichten und Objekte	Bedeutung
Gerätekontext-Objekt	Ein Gerätekontext-Objekt ist das Objekt, um das sich die GDI hauptsächlich kümmert.
BeginPaint	Die Funktion <code>BeginPaint</code> setzt alle Bereiche wieder auf gültig und liefert einen Handle auf ein Gerätekontext-Objekt zurück, das die Bereiche des Clientbereichs eines Fensters angibt, der neu gezeichnet werden sollen.
EndPaint	Zeichnet an das Ende eines Zeichenvorgangs, der durch <code>BeginPaint</code> eingeleitet wurde, einen Gerätekontext.
PAINTSTRUCT	Die Struktur vom Typ <code>PAINTSTRUCT</code> enthält Zeicheninformationen von <code>BeginPaint</code> .
GetDC	Liefert einen Handle auf ein Gerätekontext-Objekt zurück, das den gesamten Clientbereich eines Fensters angibt, der nicht überdeckt ist.
ReleaseDC	Kennzeichnet das Ende eines Zeichenvorgangs, der durch <code>GetDC</code> oder <code>GetWindowDC</code> eingeleitet wurde.
GetWindowDC	Liefert einen Handle auf einen Gerätekontext zurück, das den gesamten Bereich eines Fensters angibt, der nicht von anderen Fenstern überdeckt ist.

Funktionen, Strukturen, Nachrichten und Objekte	Bedeutung
SetPixel	Die Funktion <code>SetPixel</code> setzt einen Pixel in einem Gerätekontext auf eine bestimmte Farbe.
GetPixel	Die Funktion <code>GetPixel</code> liefert den Farbwert eines Pixels zurück.
MoveToEx	Setzt den Zeichenpunkt in einem Gerätekontext auf eine bestimmte Position.
POINT	Gibt die Position eines Punktes an.
LineTo	Die Funktion <code>LineTo</code> zeichnet eine Linie von dem eingestellten Zeichenpunkt zu einem angegebenen Punkt mit dem gesetzten Pen-Objekt.
PolyLine	Die Funktion <code>PolyLine</code> zeichnet eine Reihe von Linien, die sich mit dem Pen-Objekt verbinden.
PolyBezier	Die Funktion <code>PolyBezier</code> zeichnet mit dem Pen-Objekt eine Bezierkurve.
Rectangle	Die Funktion <code>Rectangle</code> zeichnet ein Rechteck. Für die Randfarbe wird das Pen-Objekt und für die Hintergrundfarbe das Brush-Objekt genutzt.
FillRect	Die Funktion <code>FillRect</code> zeichnet mit dem Brush-Objekt ein ausgefülltes Rechteck.
RECT	Die <code>RECT</code> Struktur beschreibt die Punkte eines Rechtecks.
Ellipse	Die Funktion <code>Ellipse</code> zeichnet eine Ellipse. Der Rand wird mit dem Pen-Objekt und der Hintergrund mit dem Brush-Objekt gezeichnet.
CreatePen	Die Funktion <code>CreatePen</code> erstellt ein Pen-Objekt.

Funktionen, Strukturen, Nachrichten und Objekte	Bedeutung
Pen-Objekt	Das Pen-Objekt ist ein Objekt, das mit einem Gerätekontext verbunden wird und meistens die Randfarbe für eine Zeichenfunktion bestimmt.
SelectObject	Die Funktion <code>SelectObject</code> verbindet einen Gerätekontext mit einem bestimmten GDI-Objekt, wie einem Pen-Objekt, einem Brush-Objekt oder einem Region-Objekt.
DeleteObject	Die Funktion <code>DeleteObject</code> löscht ein bestimmtes GDI-Objekt, wie ein Pen-Objekt, ein Brush-Objekt oder ein Region-Objekt.
CreateSolidBrush	Die Funktion <code>CreateSolidBrush</code> erstellt ein Brush-Objekt mit einer bestimmten Farbe.
Brush-Objekt	Das Brush-Objekt ist ein Objekt, das mit einem Gerätekontext verbunden ist und die Hintergrundfarbe bei einer Zeichenfunktion festlegt.
TextOut	Die Funktion <code>TextOut</code> schreibt einen String an eine bestimmte Stelle.
SetTextColor	Die Funktion <code>SetTextColor</code> setzt die Textfarbe eines Gerätekontexts.
SetBkColor	Die Funktion <code>SetBkColor</code> setzt die Hintergrundfarbe eines Gerätekontexts. Diese wird auch dazu verwendet, den Hintergrund von <code>TextOut</code> zu setzen.
SetTextAlign	Die Funktion <code>SetTextAlign</code> bestimmt die Anordnung des Textes für einen Gerätekontext.
SetBkMode	Die Funktion <code>SetBkMode</code> bestimmt eine zusätzliche Eigenschaft für die Hintergrundfarbe.

Funktionen, Strukturen, Nachrichten und Objekte	Bedeutung
RGB	Das Makro <code>RGB</code> macht aus Farbanteilen eine Farbe.
<code>CreateRectRgn</code>	Die Funktion <code>CreateRectRgn</code> erstellt ein Region-Objekt, das eine rechteckige Form hat.
Region-Objekt	Das Region-Objekt ist ein Objekt, das einen Gerätekontext für einen bestimmten Bereich angibt, in den nur gezeichnet werden darf.
<code>CombineRgn</code>	Die Funktion <code>CombineRegion</code> kombiniert Regionen auf verschiedene Weisen.
<code>SetWindowRgn</code>	Die Funktion <code>SetWindowRgn</code> weist einem bestimmten Fenster eine Region zu. Diese Region wird dann an den Gerätekontext übergeben, der von diesem Fenster ermittelt wird.
<code>GetWindowRgn</code>	Die Funktion <code>GetWindowRgn</code> ermittelt die Region eines Fensters.
<code>GetStockObject</code>	Die Funktion <code>GetStockObject</code> liefert einen Handle auf ein vordefiniertes Objekt.
<code>DrawText</code>	Die Funktion <code>DrawText</code> gibt Text auf dem Bildschirm aus.

*Tabelle 17.1: Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die GDI*

### 17.1.1 Gerätekontext-Objekt

Das Gerätekontext-Objekt ist das wichtigste Objekt der GDI. Fast alle anderen Objekte werden mit ihm verknüpft. Das Gerätekontext-Objekt ist das Objekt, in das mit GDI-Funktionen gezeichnet werden kann. Es beschreibt einen Speicherbereich des grafischen Ausgabegerätes. Über Regionen werden dem Objekt die Bereiche zugewiesen, in denen die Zeichnungen wirksam werden.

### 17.1.2 BeginPaint

Die Funktion `BeginPaint` setzt alle Bereiche eines Gerätekontexts wieder auf gültig. Das heißt es gibt keinen Bereich, der neu gezeichnet werden muss. Dadurch werden keine `WM_PAINT`-Nachrichten mehr ausgelöst. Alle Zeichen werden nur in der Region sichtbar, die wieder für gültig erklärt wurde. Die Region des Gerätekontexts wird also weiter verkleinert. Das ist der Grund, warum diese Funktion hauptsächlich in Verbindung mit der `WM_PAINT`-Nachricht eingesetzt wird. Weiter liefert die Funktion einen Handle auf ein Gerätekontext-Objekt zurück - wenn eines existiert. Wenn das Fenster nicht angezeigt wird, existiert natürlich auch keines. Die Funktion liefert in einer Struktur `PAINTSTRUCT` außerdem Informationen über den Bereich, den sie selber schon gezeichnet hat. Die Funktion zeichnet nämlich die Bereiche mit dem in der `WNDCLASS` Struktur angegebenen Brush-Objekt, neu. Dies sorgt in der Nachricht `WM_PAINT` dafür, dass keine Grafikfehler gemacht werden.

```
HDC BeginPaint( HWND hwnd, LPPAINTSTRUCT lpPaint);
```

- E: `hwnd` ist der Handle auf das Fensterobjekt, von dem der Gerätekontext-Handle ermittelt werden soll.
- E: `lpPaint` ist ein Zeiger auf eine Struktur vom Typ `PAINTSTRUCT`. Diese Struktur enthält Zeicheninformationen.
- A/R: Als Rückgabewert wird der Handle auf einen Gerätekontext des Clientbereichs zurückgegeben. Dieser gibt nur die Region an, die neu gezeichnet werden soll.
- N: Falls kein Gerätekontext für das Fenster verfügbar ist, weil es gerade nicht angezeigt wird, liefert die Funktion `NULL` zurück. Mit der Funktion `GetLastError` können Fehler unter Windows NT genauer bestimmt werden.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`



### 17.1.3 EndPaint

Die Funktion `EndPaint` kennzeichnet das Ende eines Zeichenvorgangs in ein Gerätekontext-Objekt. Diese Funktion muss nach `BeginPaint` aufgerufen werden.

```
BOOL EndPaint( HWND hWnd, CONST PAINTSTRUCT *lpPaint);
```

- E: `hWnd` ist der Handle auf das Fenster, dessen Gerätekontext angesprochen werden soll.
- E: `lpPaint` ist ein Zeiger auf die Struktur vom Typ `PAINTSTRUCT`, die von der Funktion `BeginPaint` zurückgeliefert wird.
- A/R: Als Rückgabewert wird immer ungleich `NULL` zurückgeliefert.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

### 17.1.4 PAINTSTRUCT

Die Struktur vom Typ `PAINTSTRUCT` enthält Zeicheninformationen, die von der Funktion `BeginPaint` geliefert werden.

```
typedef struct tagPAINTSTRUCT
{
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

- A: `hdc` enthält den Handle auf ein Gerätekontext, in den gezeichnet werden sollte.

- **A:** `fErase` bestimmt, ob der Hintergrund neu gezeichnet werden muss, oder ob dies bereits von `BeginPaint` erledigt wurde. Der Hintergrund muss neu gezeichnet werden, wenn in der Fensterklasse kein `Brush`-Objekt angegeben wurde.
- `rcPaint` ist eine Struktur vom Typ `RECT`, die einen rechteckigen Bereich angibt, in den neu gezeichnet werden soll.
- `fRestore` ist reserviert, weil `fRestore` intern genutzt wird.
- `fInvalidate` ist reserviert, weil `fInvalidate` intern genutzt wird.
- `rgbReserved[32]` ist reserviert, weil `rgbReserved[32]` intern genutzt wird.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

17

### 17.1.5 GetDC

Die Funktion `GetDC` liefert den gesamten Clientbereich eines Fensters zurück, der nicht von anderen Fenstern überdeckt wird. Im Gegensatz zu `BeginPaint` liefert diese Funktion wirklich einen Handle auf ein Objekt, das den gesamten freien Clientbereich angibt.

`HDC GetDC( HWND hWnd );`

- **E:** `hWnd` gibt den Handle des Fensters an, dessen Clientbereich zurückgeliefert werden soll.  
**N:** Falls dieser Wert `NULL` ist, liefert die Funktion einen Gerätekontext für den gesamten Bildschirm zurück. Dies ist nur unter Windows 98, Windows NT 5.0 oder unter späteren Versionen möglich.
- **A/R:** Als `return` wird der Handle auf einen Gerätekontext zurückgegeben.
- **N:** Falls kein Gerätekontext für das Fenster existiert, weil das Fenster nicht angezeigt wird, wird der Wert `NULL` zurückgeliefert. Mit der Funktion `GetLastError` können Fehler unter Windows NT genauer bestimmt werden.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

### 17.1.6 ReleaseDC

Die Funktion `ReleaseDC` kennzeichnet das Ende eines Zeichenvorgangs, der durch die Funktion `GetDC` oder `GetWindowDC` eingeleitet wurde.

```
int ReleaseDC( HWND hWnd, HDC hDC);
```

- E: `hWnd` ist der Handle auf ein Fensterobjekt, dessen Gerätekontext wieder freigegeben werden soll.
- E: `hDC` ist der Handle auf den Gerätekontext des Fensters, das durch `hWnd` angegeben wurde.
- A/R: Wenn der Gerätekontext freigegeben wurde, liefert die Funktion 1 zurück.
- N: Falls der Gerätekontext nicht freigegeben wurde, liefert die Funktion `NULL` zurück. Mit der Funktion `GetLastError` können Fehler unter Windows NT genauer bestimmt werden.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

### 17.1.7 GetWindowDC

Die Funktion `GetWindowDC` liefert einen Handle auf ein Gerätekontext-Objekt zurück, das sich auf die ganze Fläche, nicht nur auf den Clientbereich, des Fensters bezieht, die nicht überdeckt wird.

HDC GetWindowDC( HWND hWnd);

- E: `hWnd` ist der Handle auf ein Fenster, dessen Gerätekontext ermittelt werden soll. Dieser Gerätekontext bezieht sich auf das gesamte Fenster und nicht nur auf den Teilbereich Client.
- N: Falls dieser Wert `NULL` ist, liefert die Funktion `GetWindowDC` einen Gerätekontext für den gesamten Bildschirm ohne Einschränkungen zurück. Diese Funktion ist nur unter Windows 98, Windows NT 5.0 oder späteren Versionen einsetzbar.
- A/R: Als `return` liefert die Funktion `GetWindowDC` einen Handle auf ein Gerätekontext zurück, der sich auf das gesamte Fenster bezieht.

N: Falls ein Fehler auftritt, liefert die Funktion den Wert `NULL` zurück. Mit der Funktion `GetLastError` können Fehler unter Windows NT genauer bestimmt werden.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

### 17.1.8 SetPixel

Die Funktion `SetPixel` setzt einen Pixel eines Gerätekontexts auf einen bestimmten Farbwert. Falls dieser nicht vorhanden ist, wird der Wert ermittelt, der dem gewünschten Farbwert am meisten gleicht.

```
COLORREF SetPixel(HDC hdc, int X, int Y,  
                  COLORREF crColor);
```

- `hdc` ist der Handle auf das Gerätekontext-Objekt, in das gezeichnet werden soll.
- `X`, `Y` sind relative, logische Angaben, die sich auf die obere linke Ecke des Gerätekontexts beziehen.
- `crColor` ist der Farbwert, auf den der Pixel gesetzt werden soll.

- A/R: Wenn die Funktion den Pixel auf einen Farbwert setzt, liefert sie diesen Farbwert zurück. Dieser kann von dem gewünschten Farbwert variieren, weil dieser ggf. angepasst wird, wenn er nicht dargestellt werden kann.
- N: Falls ein Fehler auftritt, liefert die Funktion `-1` zurück. Mit der Funktion `GetLastError` können Fehler unter Windows NT genauer bestimmt werden.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.9 GetPixel

Die Funktion `GetPixel` liefert den Farbwert eines Pixels eines Gerätekontexts zurück. Dabei wird seine Position angegeben.

```
COLORREF GetPixel( HDC hdc, int XPos, int YPos);
```

- `hdc` ist der Handle auf das Gerätekontext-Objekt, von dem die Farbwerte eines Pixels ermittelt werden sollen.
- `XPos` und `YPos` geben die Position relativ zur obersten linken Ecke des Gerätekontext-Objekts an.
- A/R: Als `return` wird der Farbwert des Pixels an der entsprechenden Stelle zurückgegeben. Falls der gewünschte Farbwert außerhalb der Region liegt, aus der man Farbwerte ermitteln darf, wird der Wert von einer Konstanten zurückgegeben.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.10 MoveToEx

Die Funktion `MoveToEx` bestimmt den Zeichenpunkt. Jeder Gerätekontext hat einen solchen Zeichenpunkt. Dieser wird relativ zur oberen linken Ecke des Gerätekontexts angegeben. Sein Wert kann mit der Funktion `MoveToEx` festgesetzt werden. Dabei werden die relativen X- und Y- Koordinaten angegeben. Die Position wird dann von Zeichenfunktionen genutzt.

```
BOOL MoveToEx( HDC hdc, int X, int Y, LPPOINT lpPOINT);
```

- E: `hdc` ist der Handle auf einen Gerätekontext. In diesem Gerätekontext wird der Zeichenpunkt gesetzt.
- E: `X` und `Y` sind die relativen, logischen Positionen von der oberen linken Ecke des Gerätekontexts aus. Diese bestimmen den Zeichenpunkt.
- A: `lpPOINT` ist ein Zeiger auf eine Struktur. In dieser Struktur werden Daten über die Position des Zeichenpunkts zurückgeliefert.
- A/R: Wenn die Funktion ohne Fehler ausgeführt wird, liefert die Funktion ungleich `NULL` zurück.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion `NULL` zurück. Unter Windows NT kann man mit der Funktion `GetLastError` genauere Informationen über den Fehler erhalten.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE nicht unterstützt

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.11 POINT

Die Struktur `POINT` enthält Angaben über einen Punkt. Dieser Punkt wird durch logische X- und Y- Werte beschrieben.

```
typedef struct tagPOINT
{
    LONG x;
    LONG y;
} POINT;
```

- E/A:  $x$  und  $y$  sind Positionsangaben. Diese Angaben sind logische Angaben, mit denen man einen Punkt von einer oberen linken Ecke logisch beschreiben kann. Meistens handelt es sich dabei um Pixelangaben.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINDEF.H

### 17.1.12 LineTo

Die Funktion `LineTo` zeichnet durch `MoveToEx` eine Linie vom angegebenen Zeichenpunkt zu einem bestimmten Punkt. `LineTo` benutzt die Einstellungen des Pen-Objekts, um seiner Linie eine Farbe und weitere Eigenschaften zu geben.

```
BOOL LineTo( HDC hdc, int nXEnd, int nYEnd);
```

- E: `hdc` ist der Handle auf den Gerätekontext, in den die Linie gezeichnet werden soll.
- E: `nXEnd` und `nYEnd` sind Angaben für einen Punkt relativ zu der oberen linken Ecke des Gerätekontexts, zu dem die Linie gezeichnet wird. Die Linie geht von dem Punkt aus, der durch `MoveToEx` angegeben wird.
- A/R: Falls die Funktion ohne Fehler ausgeführt wird, liefert sie einen Wert ungleich `NULL` zurück.
- N: Falls aber doch ein Fehler bei der Ausführung der Funktion auftritt, liefert die Funktion den Wert `NULL` zurück. Um genauere Informationen über den Fehler zu erhalten, ruft man unter Windows NT die Funktion `GetLastError` auf.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE nicht unterstützt

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.13 PolyLine

Die Funktion `PolyLine` zeichnet eine Reihe von zusammenhängenden Linien. Die Punkte der Linien werden durch einen Array vom Typ `POINT` angegeben.

```
BOOL PolyLine( HDC hdc, CONST POINT *lppt, int cPOINTS);
```

- E: `hdc` ist der Handle auf einen Gerätekontext, in den die Linienfolge gezeichnet werden soll.
- E: `lppt` ist der Zeiger auf einen Array von `POINT`-Strukturen, die zur Eingabe genutzt werden. Die Eingabe besteht aus den Punkten, aus denen die Linienfolge besteht. Die Anzahl der Punkte muss mindestens zwei betragen.
- E: `cPOINTS` gibt die Anzahl der Punkte in der Struktur an, auf die `lppt` zeigt.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich `NULL`. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB



### 17.1.14 PolyBezier

Die Funktion `PolyBezier` zeichnet eine Reihenfolge von Linien im Bezierstil.

```
BOOL PolyBezier( HDC hdc, CONST POINT *lppt,
                DWORD cPOINTS);
```

- E: `hdc` ist der Handle auf ein Gerätekontext, in das die Reihenfolge der Linien gezeichnet werden soll.
- E: `lppt` ist ein Zeiger auf eine Reihe von `POINT`-Strukturen. Diese Strukturen geben die Punkte zwischen den Bezierlinien an.
- E: `cPOINTS` gibt die Anzahl der Punkte an.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich `NULL`. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE nicht unterstützt

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.15 Rectangle

Die Funktion `Rectangle` zeichnet ein Rechteck. Dabei wird das Pen-Objekt für die Randfarbe genutzt. Das Brush-Objekt wird für die Hintergrundfarbe genutzt.

```
BOOL Rectangle( HDC hdc, int nLeftRect, int nTopRect,
                int nRightRect, int nBottomRect);
```

- E: `hdc` ist ein Handle auf ein Gerätekontext-Objekt, in das das Rechteck gezeichnet werden soll.

- E: `nLeftRect`, `nTopRect`, `nRightRect` und `nBottomRect` geben eine relative, logische Position für das Rechteck an.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich `NULL`. Wenn man Genaueres über den Fehler wissen möchte, muss man auch hier die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.16 FillRect

Die Funktion `FillRect` zeichnet ein ausgefülltes Rechteck. Die Funktion verwendet dazu das gesetzte Brush-Objekt.

```
int FillRect( HDC hdc, CONST RECT *lprc, HBRUSH hbr);
```

- E: `hdc` ist der Handle auf einen Gerätekontext. In diesen Gerätekontext wird das ausgefüllte Rechteck gezeichnet. Die Funktion hat eine Besonderheit: Die obere linke Ecke schließt die Kante ein, während die untere rechte Ecke die Kante ausschließt.
- E: `lprc` ist ein Zeiger auf eine `RECT`-Struktur. Diese `RECT`-Struktur enthält die Angaben über die Position und Maße des Rechtecks.
- E: `hbr` ist das Brush-Objekt mit dem das Rechteck gezeichnet wird.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich `NULL`. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

17

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

### 17.1.17 RECT

Die Struktur RECT beschreibt die Angaben von einem Rechteck. Zwei Punkte werden über Koordinaten beschrieben, wobei zwischen diesen Punkten dann das Rechteck aufgespannt wird.

```
typedef struct _RECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

- E/A: `left`, `top`, `right` und `bottom` geben zwei Punkte an, zwischen denen das Rechteck aufgespannt wird.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINDEF.H

### 17.1.18 Ellipse

Die Funktion `Ellipse` zeichnet eine Ellipse. Für den Rand der Ellipse wird das Pen-Objekt genutzt, das gerade gesetzt ist, ihr Hintergrund wird mit dem gerade belegten Brush-Objekt ausgefüllt.

```
BOOL Ellipse( HDC hdc, int nLeftRect, int nTopRect,
              int nRightRect, int nBottomRect);
```

- `hdc` ist der Handle auf den Gerätekontext, in den die Ellipse gezeichnet wird.
- `nLeftRect`, `nTopRect`, `nRightRect`, `nBottomRect` sind die Angaben für zwei Punkte, die ein Rechteck aufspannen. In dieses Rechteck wird die Ellipse gezeichnet.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich `NULL`. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

17

### 17.1.19 CreatePen

Die Funktion `CreatePen` erstellt ein Pen-Objekt. Das Pen-Objekt wird bei bestimmten Zeichenfunktionen meistens für den Rand genutzt. Zu diesem Zweck wird es über die Funktion `SelectObject` mit dem Gerätekontext verbunden.

```
HPEN CreatePen( int fnPenStyle, int nWidth,
                COLORREF crColor);
```

- E: `fnPenStyle` wird durch eine Konstante bestimmt. Die Konstanten können nur einzeln auftreten.

`PS_SOLID` erstellt ein normales Pen-Objekt. Mit diesem Pen-Objekt können ganze Linien gezeichnet werden.

`PS_DASH` erstellt ein Pen-Objekt, das Striche statt einer ausgefüllten Linie zeichnet. Dies ist nur möglich, wenn `nWidth` eins oder kleiner als eins ist.

PS\_DOT erstellt ein Pen-Objekt, das Striche zeichnet, die aus Punkten bestehen. Dies ist nur möglich, wenn `nWidth` eins oder kleiner als eins ist.

PS\_DASHDOT bestimmt, dass die Linie, die gezeichnet wird, aus Strichen und Punkten besteht. Dies ist nur möglich, wenn `nWidth` eins oder kleiner als eins ist.

PS\_DASHDOTDOT bewirkt, dass die Linie, die gezeichnet wird, aus folgender, sich wiederholender Reihenfolge besteht: Strich, Punkt und Punkt. Dies ist nur möglich, wenn `nWidth` eins oder kleiner als eins ist.

PS\_NULL bestimmt, dass die Linie nicht gezeichnet wird.

- E: `nWidth` legt die Breite der Linien fest.
- E: `crColor` definiert den Farbwert des Pen-Objekts, mit dem die Linien gezeichnet werden.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich NULL, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich NULL. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 2.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.20 Pen-Objekt

Das Pen-Objekt enthält Angaben über eine Linie, die sich auf die Breite der Linie, ihr Aussehen und ihre Farbe beziehen. Sie gelten für alle logischen Pen-Objekte, die mit der Funktion `CreatePen` erzeugt werden können. Diese Pen-Objekte werden dazu genutzt, Linien und Kurven zu zeichnen.

### 17.1.21 SelectObject

Die Funktion `SelectObject` verbindet einen Pen, Brush, Font, Bitmap und eine Region mit einem Gerätekontext. Falls eine Region mit dem Gerätekontext verbunden werden soll, ist diese Region die Update-Region. Zusammen bilden die Update- und die Visible-Region die Clipping-Region. Die Clipping-Region ist der Bereich, in dem die Zeichenfunktionen wirksam werden.

```
HGDIOBJ SelectObject( HDC hdc, HGDIOBJ hgdioobj);
```

- E: `hdc` ist der Handle auf einen Gerätekontext. Mit diesem Gerätekontext werden die Objekte, die in `hgdioobj` angegeben sind, verbunden. Der Gerätekontext erhält Kenntnis von dem anderen Objekt.
- E: `hgdioobj` ist der Handle auf das Objekt, das dem Gerätekontext zugewiesen wird.
- A/R: Falls die Funktion ohne Fehler ausgeführt wurde und das Objekt von `hgdi` keine Region ist, gibt sie den Handle auf das alte Objekt zurück, das nun nicht mehr mit dem Gerätekontext verbunden ist. Falls das Objekt aber eine Region ist, gibt die Funktion den Wert einer Konstanten zurück.

`SIMPLEREGION` bedeutet, dass die neue Region aus einem Rechteck besteht.

`COMPLEXREGION` besagt, dass die neue Region aus mehreren Rechtecken besteht.

`NULLREGION` bedeutet, dass die Region keine Größe hat.

Falls ein Error auftritt und das Objekt keine Region ist, liefert die Funktion den Wert `NULL` zurück, andernfalls den Wert von der Konstanten `GDI_ERROR`.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.22 DeleteObject

Die Funktion `DeleteObject` löscht ein Objekt. Dabei können die Objekte vom Typ logisches Pen-Objekt, Brush-Objekt, Font-Objekt, Bitmap-Objekt, Region-Objekt oder Paletten-Objekt sein.

```
BOOL DeleteObject( HGDIOBJ hObject);
```

- E: `hObject` ist der Handle auf das Objekt, das gelöscht werden soll.
- A/R: Als Rückgabewert liefert die Funktion den Wert `NULL`, falls kein Fehler aufgetreten ist.
- N: Falls der angegebene Handle aber nicht verfügbar oder in einen Gerätekontext eingebunden ist, ist der zurückgelieferte Wert `NULL`. Um den Fehler genauer zu bestimmen, kann man unter Windows NT die Funktion `GetLastError` aufrufen.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.23 CreateSolidBrush

Die Funktion `CreateSolidBrush` erstellt ein Brush-Objekt, das nur eine Farbe enthält. Brush-Objekte werden meistens für Hintergründe in Zeichenfunktionen verwendet.

```
HBRUSH CreateSolidBrush( COLORREF crColor);
```

- `crColor` ist ein RGB-Farbwert für das Brush-Objekt.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich `NULL`. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.24 Brush-Objekt

Ein Brush-Objekt enthält Angaben über eine Farbe, ein Muster oder ein Bitmap für dieses Muster. Mit Brush-Objekten wird oft ein Hintergrund mit Zeichenfunktionen erstellt. Ein Brush-Objekt, das Angaben über eine Farbe enthält, erzeugt man z.B. mit der Funktion `CreateSolidBrush`.

### 17.1.25 TextOut

Die Funktion `TextOut` gibt einen Zeichenstring auf einem Gerätekontext aus. Die Funktion nutzt den eingestellten Font für das Gerätekontext-Objekt. Außerdem verwendet sie noch andere Einstellungen des Gerätekontext-Objekts. Dazu gehören die Textausrichtung, die Textfarbe und der Texthintergrund.

```
BOOL TextOut( HDC hdc, int nXStart, int nYStart,  
             LPCTSTR lpString, int cbString);
```

- E: `hdc` ist der Handle auf den Gerätekontext, in den geschrieben werden soll.
- E: `nXStart` und `nYStart` geben die Position an, wohin der Text geschrieben werden soll. Diese Position kann je nach Textausrichtung anders interpretiert werden. Falls der Text links ausgerichtet ist, beschreibt diese Position die obere linke Ecke des Textanfangs. Falls er zentriert ist, wird der Text um diese Position symmetrisch angeordnet.
- E: `lpString` ist der Text, der ausgegeben werden soll.
- E: `cbString` ist die Länge des Texts. Dieser Wert gibt die Anzahl der Zeichen innerhalb des Texts an. Das heißt der abschließende NULL-Character ist hier nicht mit eingeschlossen. Dieser Wert wird normalerweise mit der Funktion `lstrlen` ermittelt.



- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich NULL, wenn die Funktion ohne Fehler ausgeführt wurde.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert ungleich NULL. Wenn man Genaueres über den Fehler wissen möchte, muss man die Funktion `GetLastError` aufrufen. Dies funktioniert aber nur unter Windows NT.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE nicht unterstützt

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

Unterstützung für ANSI und UNICODE unter Windows und Windows NT

### 17.1.26 SetTextColor

Die Funktion `SetTextColor` bestimmt die Textfarbe für einen bestimmten Gerätekontext.

```
COLORREF SetTextColor( HDC hdc, COLORREF crColor);
```

- E: `hdc` ist der Handle auf einen Gerätekontext. In diesem Gerätekontext wird der Wert für die Textfarbe geändert.
- E: `crColor` ist ein RGB-Farbwert für die Textfarbe.
- A/R: Wenn die Funktion ohne Fehler ausgeführt wurde, liefert sie den Wert der vorherigen Textfarbe zurück.
- N: Falls bei der Ausführung der Funktion ein Fehler auftrat, liefert die Funktion den Wert der Konstanten `CLR_INVALID` zurück. Mit der Funktion `GetLastError` wird der Fehler unter Windows NT genauer bestimmt.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.27 SetBkColor

Die Funktion `SetBkColor` bestimmt die Hintergrundfarbe eines Gerätekontexts. Falls diese Hintergrundfarbe nicht in diesem Gerätekontext vorkommt, wird sie auf die Farbe gesetzt, die ihr am ähnlichsten ist. Diese Hintergrundfarbe wird für Textausgaben und Hintergründe von Zeichenfunktionen verwendet.

```
COLORREF SetBkColor( HDC hdc, COLORREF crColor);
```

- E: `hdc` ist der Handle auf einen Gerätekontext. Die Hintergrundfarbe dieses Gerätekontexts wird ausgewählt.
- E: `crColor` ist der Farbwert, der als Hintergrundfarbe gesetzt wird.
- A/R: Wenn die Funktion ohne Fehler ausgeführt worden ist, liefert sie den Wert der vorherigen Hintergrundfarbe zurück.
- N: Falls bei der Ausführung der Funktion ein Fehler auftrat, liefert die Funktion den Wert der Konstanten `CLR_INVALID` zurück. Mit der Funktion `GetLastError` wird der Fehler unter Windows NT genauer bestimmt.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.28 SetTextAlign

Die Funktion `SetTextAlign` setzt die Ausrichtung des Texts für einen Gerätekontext. Natürlich wird nur die zukünftige Ausrichtung des Texts festgelegt.

```
UINT SetTextAlign( HDC hdc, UINT fMode);
```

17

Nitty Gritty • Take that!

- `hdc` ist der Handle auf einen Gerätekontext. Von diesem Gerätekontext wird die Textausrichtung gesetzt.
- `fMode` bestimmt, welche Ausrichtung der Text haben soll. Dabei kann eine der folgenden Konstanten angegeben werden. Zusätzlich können noch die Konstanten angegeben werden, die sich auf das Verändern der aktuellen Position beziehen. Die folgenden Angaben gelten für ein angegebenes Rechteck. Dieses Rechteck wird von der Funktion `DrawText` verwendet. Die Funktion `TextOut` gibt ja nur den Punkt an, dort werden die Funktionen entsprechend anders interpretiert.

`TA_BASELINE` bestimmt, dass der Beziehungspunkt die Basislinie ist.

`TA_BOTTOM` bewirkt, dass der Beziehungspunkt am unteren Rand des angegebenen Rechtecks liegt.

`TA_TOP` legt fest, dass der Beziehungspunkt am oberen Rand des angegebenen Rechtecks liegt.

`TA_CENTER` setzt fest, dass der Beziehungspunkt in der horizontalen Mitte des angegebenen Rechtecks liegt. Bei der Funktion `TextOut` werden die Buchstaben um den angegebenen Punkt herum angeordnet.

`TA_LEFT` bestimmt, dass der Beziehungspunkt die angegebene linke Ecke des angegebenen Rechtecks ist. Die Position, die von `TextOut` bestimmt wurde, ist die Position, an die sich der Text links anschließt.

`TA_RIGHT` sagt aus, dass der Beziehungspunkt die angegebene rechte Ecke des angegebenen Rechtecks ist. An die festgelegte Position von `TextOut` schließt sich der Text in rechter Richtung an.

`TA_NOUPDATECP` bewirkt, dass sich die aktuelle Position nicht verändert.

`TA_UPDATECP` legt fest, dass die aktuelle Position verändert wird. Außerdem wird sie als Beziehungspunkt genutzt.

- **A/R:** Wenn die Funktion ohne Fehler ausgeführt worden ist, liefert sie den Wert der vorherigen Textausrichtung zurück.

- N: Falls bei der Ausführung der Funktion ein Fehler auftrat, liefert sie den Wert der Konstanten `GDI_ERROR` zurück. Mit der Funktion `GetLastError` wird der Fehler unter Windows NT genauer bestimmt.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE nicht unterstützt

Header-Datei: `WINGDI.H`

Library-Datei: `GDI32.LIB`

### 17.1.29 SetBkMode

Die Funktion `SetBkMode` bestimmt eine zusätzliche Eigenschaft der Hintergrundfarbe.

```
int SetBkMode( HDC hdc, int iBkMode);
```

- E: `hdc` ist der Handle auf einen Gerätekontext. Zusätzlich wird die Einstellung für die Hintergrundfarbe dieses Gerätekontexts gesetzt.
- E: `iBkMode` ist der Wert einer Konstanten. Für diese Konstanten kann eine der folgenden Konstanten angegeben werden:

`OPAQUE` bestimmt, dass der Hintergrund mit der Hintergrundfarbe gefüllt wird, bevor der Hintergrund beschrieben wird.

`TRANSPARENT` bedeutet, dass der Hintergrund nicht ausgefüllt wird.

- A/R: Wenn die Funktion ohne Fehler ausgeführt wurde, liefert sie den Wert der vorherigen zusätzlichen Eigenschaft zur Hintergrundfarbe zurück.
- N: Falls bei der Ausführung der Funktion ein Fehler auftrat, liefert die Funktion den Wert 0 zurück. Mit der Funktion `GetLastError` wird der Fehler unter Windows NT genauer bestimmt.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später

17

- Windows CE 1.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.30 RGB

Das Makro RGB bestimmt aus drei Farbanteilen den Wert einer Farbe des Typs COLORREF.

```
COLORREF RGB( BYTE bRed, BYTE bGreen, BYTE bBlue);
```

- E: bRed, bGreen und bBlue geben den Wert des Farbanteils an.
- A/R: Die Funktion liefert den entstandenen Farbwert zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINGDI.H

### 17.1.31 CreateRectRgn

Die Funktion CreateRectRgn erstellt ein Region-Objekt.

```
HRGN CreateRectRgn( int nLeftRect, int nTopRect,  
                   int nRightRect, int nBottomRect);
```

- E: nLeftRect, nTopRect, nRightRect und nBottomRect bestimmen die Größe des Rechtecks. Das Rechteck wird relativ angegeben.
- A/R: Als Rückgabewert liefert die Funktion ein Region-Objekt.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, wird der Wert NULL zurück geliefert. Mit der Funktion GetLastError wird unter Windows NT Genaueres über den Fehler ermittelt.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 2.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.32 Region-Objekt

Das Region enthält Angaben über einen bestimmten Bereich. Es gibt zwei Arten von Regionen, die definiert werden können. Die eine ist die Visible-Region, die dem Fenster zugewiesen wird. Sie wird von der Größe des Fensters und von den überdeckten Bereichen bestimmt. Zusätzlich kann man diese Region noch durch die Angabe von `SetWindowRgn` einengen. Dann gibt es eine weitere Region, die dem Gerätekontext zugewiesen wird. Dabei handelt es sich um die Update-Region. Aus beiden Regionen wird der Schnittpunkt gebildet: die Clipping-Region. Jede Zeichenfunktion ermittelt die Clipping-Region neu, um nicht den Bereich anderer Anwendungen zu beeinflussen.

### 17.1.33 CombineRgn

Mit der Funktion `CombineRgn` werden Region-Objekte auf verschiedene Weise zu neuen Region-Objekten kombiniert.

```
int CombineRgn( HRGN hrgnDest, HRGN hrgnSrc1,
                HRGN hrgnSrc2, int fnCombineMode);
```

- E: `hrgnDest` ist der Handle auf die Zielregion. Diese Region muss existieren, bevor die Funktion `CombineRgn` aufgerufen wird.
- E: `hrgnSrc1` und `hrgnSrc2` sind Handles auf Regionen. Aus diesen beiden Regionen wird die Zielregion gebildet.
- E: `fnCombineMode` bestimmt, wie die beiden Regionen kombiniert werden. Dieser Wert kann eine der folgenden Konstanten sein:

`RGN_AND` bestimmt, dass die entstehende Region das logische AND aus den beiden angegebenen Regionen sein soll.

`RGN_OR` legt fest, dass die entstehende Region das logische OR aus den beiden angegebenen Regionen sein soll.

`RGN_XOR` bestimmt, dass die entstehende Region das logische XOR aus den beiden angegebenen Regionen sein soll.

`RGN_COPY` besagt, dass die Region `hrgnSrc1` kopiert werden soll.

17

Nitty Gritty • Take that!

- A/R: Als Rückgabewert liefert die Funktion den Wert einer Konstanten. Die Werte folgender Konstanten sind möglich:

NULLREGION bestimmt, dass die entstandene Region leer ist.

SIMPLEREGION bewirkt, dass die entstandene Region ein einfaches Rechteck ist.

COMPLEXREGION legt fest, dass die entstandene Region mehr als ein einfaches Rechteck ist.

ERROR bestimmt, dass keine Region erstellt worden ist, weil ein Fehler aufgetreten ist.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.34 SetWindowRgn

Die Funktion `SetWindowRgn` bestimmt eine Region für ein bestimmtes Fenster. Diese Region wird mit der Region kombiniert, die nicht vom Fenster überdeckt wird: die Visible-Region.

```
int SetWindowRgn( HWND hWnd, HRGN hRgn, BOOL bRedraw);
```

- `hWnd` ist der Handle auf das Fenster, von dem die Region gesetzt werden soll.
- `hRgn` ist der Handle auf die Region, die gesetzt werden soll.
- `bRedraw` bestimmt, dass das Fenster neu gezeichnet werden soll, nachdem die Region gesetzt worden ist. Bei dem Wert TRUE wird das Fenster neu gezeichnet. Bei dem Wert FALSE wird das Fenster nicht neu gezeichnet.
- A/R: Wenn die Funktion ohne Fehler ausgeführt wurde, liefert sie einen Wert ungleich NULL zurück.

- N: Falls die Funktion mit Fehlern ausgeführt wurde, liefert die Funktion den Wert NULL zurück. Die Funktion `GetLastError` liefert unter Windows NT genauere Informationen über den Fehler.

Unterstützung:

- Windows NT 3.51 und später
- Windows 95 und später
- Windows CE nicht unterstützt

Header-Datei: `WINUSER.H`

Library-Datei: `USER32.LIB`

### 17.1.35 **GetStockObject**

Die Funktion `GetStockObject` liefert den Handle für ein vordefiniertes Objekt und von Objekten wie Pen, Brush, Fonts und Paletten-Objekten.

`HGDIOBJ GetStockObject( int fnObject);`

- E: `fnObject` ist der Wert einer Konstanten. Diese Konstante gibt an, welcher Handle von welchem Objekt zurückgeliefert werden soll.

`WHITE_BRUSH` bestimmt, dass ein Handle auf ein weißes Brush-Objekt zurückgeliefert wird.

`BLACK_BRUSH` bewirkt, dass ein Handle auf ein schwarzes Brush-Objekt zurückgeliefert wird.

`DKGRAY_BRUSH` legt fest, dass ein Handle auf ein dunkelgraues Brush-Objekt zurückgeliefert wird.

`GRAY_BRUSH` besagt, dass ein Handle auf ein graues Brush-Objekt zurückgeliefert wird.

`LTGRAY_BRUSH` bestimmt, dass ein Handle auf ein hellgraues Brush-Objekt zurückgeliefert wird.

`DC_BRUSH` setzt fest, dass ein Handle auf ein definiertes Brush-Objekt zurückgeliefert wird. Die Standardfarbe ist Weiß. Die Farbe kann mit der Funktion `SetDCBrushColor` gesetzt werden.

17



NULL\_BRUSH erreicht, dass ein Handle auf ein vordefiniertes Brush-Objekt zurückgeliefert wird. Überall, wo dieses Brush-Objekt zum Einsatz kommt, wird es nicht gezeichnet.

WHITE\_PEN bewirkt dass ein Handle auf ein weißes Pen-Objekt zurückgeliefert wird.

BLACK\_PEN bestimmt, dass ein Handle auf ein schwarzes Pen-Objekt zurückgeliefert wird.

DC\_PEN legt fest, dass ein Handle auf ein vordefiniertes Pen-Objekt zurückgeliefert wird. Dieses Pen-Objekt kann mit der Funktion SetDCPenColor gesetzt werden.

ANSI\_FIXED\_FONT erreicht, dass ein Handle auf ein Font-Objekt zurückgeliefert wird.

ANSI\_VAR\_FONT bestimmt, dass ein Handle auf ein Font-Objekt zurückgeliefert wird. Dieser Font ist eine proportional Schrift.

OEM\_FIXED\_FONT besagt, dass ein Handle auf ein Font-Objekt zurückgeliefert wird. Dies ist der erweiterte DOS-Zeichensatz mit Blockgrafiken. Dieser Zeichensatz wird für die Abwärtskompatibilität bereitgestellt. Es handelt sich um eine nicht proportionale Schrift.

SYSTEM\_FONT legt fest, dass ein Handle auf ein Font-Objekt zurückgeliefert wird. Dieser Font wird benutzt, um Menüs und vordefinierte Dialogboxen zu zeichnen.

SYSTEM\_FIXED\_FONT bewirkt, dass ein Handle auf ein Font-Objekt zurückgeliefert wird. Dieser Font wird für Windowsversionen bereitgestellt, die älter als Windows 3.0 sind.

DEFAULT\_GUI\_FONT bestimmt, dass ein Handle auf ein Font-Objekt zurückgeliefert wird.

DEFAULT\_PALETTE besagt, dass ein Handle auf ein Paletten-Objekt zurückgeliefert wird, das die Standardssystemfarben enthält.

- A/R: Zurückgeliefert wird der Handle auf das GDI-Objekt.
- N: Falls ein Fehler bei der Ausführung der Funktion auftritt, liefert die Funktion den Wert NULL zurück. Mit der Funktion `GetLastError` werden weitere Informationen über den Fehler abgerufen.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINGDI.H

Library-Datei: GDI32.LIB

### 17.1.36 DrawText

Die Funktion `DrawText` gibt Text auf dem Bildschirm aus. Dazu wird ein Rechteck auf dem Bildschirm definiert.

```
int DrawText( HDC hDC, LPCTSTR lpString, int nCount,  
             LPRECT lpRect, UINT uFormat);
```

- E: `hDC` ist der Handle auf den Gerätekontext, in den der Text ausgegeben werden soll.
- E: `lpString` ist der Text, der ausgegeben werden soll.
- E: `nCount` ist die Anzahl der Zeichen des Textes. Dieser Wert muss `-1` sein, falls die Zeichenfolge mit einem NULL-Zeichen abgeschlossen wird. Dann ermittelt die Funktion die Zeichenfolge selber.
- E: `lpRect` ist ein Zeiger auf eine Struktur vom Typ `RECT`. Diese Struktur enthält die Daten des Rechtecks, in das der Text ausgegeben werden soll.
- E: `uFormat` bestimmt, wie der Text im Rechteck ausgegeben wird. Dabei kann jede Kombination der folgenden Konstanten benutzt werden.

`DT_BOTTOM` legt fest, dass der Text am Boden des Rechtecks ausgerichtet wird. Diese Konstante muss mit der Konstanten `DT_SINGLELINE` kombiniert werden.

`DT_CALCRECT` bewirkt, dass die Größe des Rechtecks für einen mehrzeiligen Text definiert wird.

`DT_CENTER` bestimmt, dass der Text im Rechteck horizontal zentriert ausgegeben wird.

17

DT\_EDITCONTROL erreicht, dass der Text genauso ausgegeben wird, wie in einem *Multi-Line-Edit-Control*.

DT\_END\_ELLIPSES/ DT\_PATH\_ELLIPSES legt fest, dass der Text der Größe des Rechtecks angepasst wird. Um dies zu erreichen, muss auch DT\_MODIFYSTRING angegeben werden.

DT\_EXPANDTABS erweitert die Tab-Zeichen. Die voreingestellte Länge eines Tab-Zeichens ist 8.

DT\_EXTERNAL\_LEADING bestimmt, dass der Text mit *external leading* in der Höhe angegeben wird.

DT\_INTERNAL bewirkt dass der Systemfont benutzt wird, um die Texteingstellungen zu verändern.

DT\_LEFT bestimmt, dass der Text linksausgerichtet wird.

DT\_MODIFYSTRING legt fest, dass der Text dem Ausgabebereich angepasst wird. Falls DT\_END\_ELLIPSES oder DT\_PATH\_ELLIPSES angegeben worden sind, hat diese Konstante keinen Effekt.

DT\_NOCLIP bestimmt, dass der Text ohne Clipping gezeichnet wird.

DT\_NOPREFIX bewirkt, dass ein & nicht mehr zum Unterstreichen genutzt, sondern dass es auf dem Bildschirm ausgegeben wird.

DT\_RIGHT bestimmt, dass der Text am rechten Rand des Rechtecks ausgerichtet wird.

DT\_SINGLELINE definiert, dass der Text in einer Linie angezeigt wird.

DT\_TABSTOP bestimmt, dass die Anzahl der Zeichen für einen Tab-stop gesetzt werden.

DT\_TOP setzt fest, dass der Text am oberen Rand des Rechtecks ausgerichtet wird.

DT\_VCENTERS bestimmt, dass der Text vertikal zentriert wird.

DT\_WORDBREAK erreicht, dass Wörter automatisch getrennt werden.

DT\_WORD\_ELLIPSES gibt an, dass die Wörter nicht ganz ausgeschrieben werden, wenn sie nicht in das Rechteck passen. Es wird dann eine Ellipse angefügt.

- A/R: Falls die Funktion ohne Fehler ausgeführt wird, liefert die Funktion die Höhe des Textes zurück.
- N: Falls bei der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert NULL zurück. Mit der Funktion `GetLastError` wird unter Windows NT der Fehler genauer bestimmt.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINUSER.H

Library-Datei: USER32.LIB

Unterstützung für ANSI und UNICODE unter Windows NT

17

## 17.2 Beispiele

### 17.2.1 Einen Grafikbereich in der Nachricht WM\_PAINT ermitteln und etwas hineinzeichnen

Wenn die Nachricht WM\_PAINT abgearbeitet werden muss, wird ein Gerätekontext ermittelt. Dieser beschreibt aber nur die Region, die neu gezeichnet werden muss. In diesen Gerätekontext werden dann ein Rechteck und eine Ellipse gezeichnet. Danach wird er wieder freigegeben.

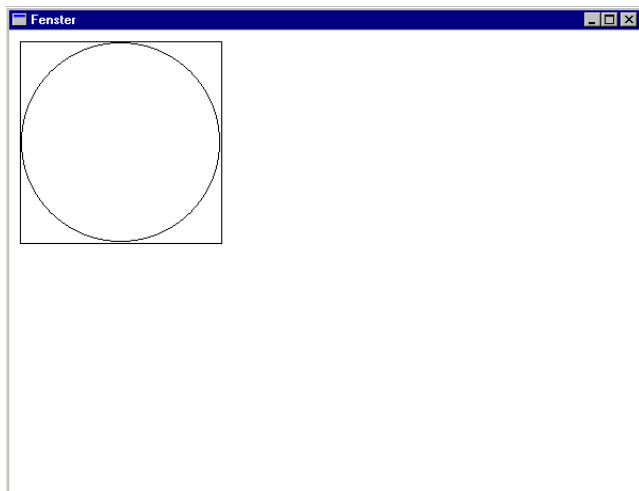


Bild 17.1: Die Anwendung nach dem Start

## Quelltext

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
```

```
WndClass.lpszClassName = "WinProg";

RegisterClass(&WndClass);

HWND hWnd;
hWnd = CreateWindow("WinProg", "Fenster",
    WS_OVERLAPPEDWINDOW,
    0, 0, 600, 460, NULL, NULL,
    hInstance, NULL);

ShowWindow (hWnd, nCmdShow);

UpdateWindow (hWnd);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
    WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_PAINT:
            HDC hdc;
            PAINTSTRUCT ps;
            hdc = BeginPaint (hWnd, &ps);

            Rectangle (hdc, 10, 10, 200, 200);

            Ellipse (hdc, 11, 11, 199, 199);

            EndPaint (hWnd, &ps);
    }
}
```

```

        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc (hWnd, uiMessage,
                               wParam, lParam);
    }
}

```

## 17.2.2 Pen- und Brush-Objekte erstellen und zuweisen

### Beschreibung

In der Nachricht WM\_PAINT werden ein Pen- und ein Brush-Objekt erstellt. Beide werden dem Gerätekontext zugewiesen, der von dem Fenster erstellt wurde. Die alten Objekte des Gerätekontexts werden gesichert und am Ende wieder zugewiesen. Durch Pen- und Brush-Objekte ist es möglich, die Zeichen- und Füllfarbe verschiedener Zeichenfunktionen zu verändern.

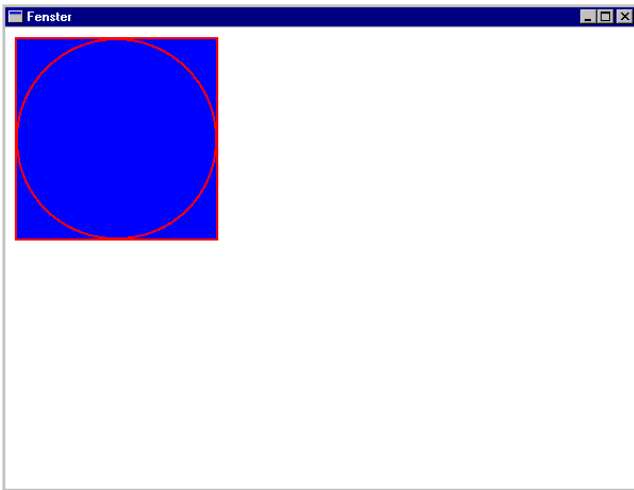


Bild 17.2: Die Anwendung nach dem Start

## Quelltext

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )

{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
    hWindow = CreateWindow("WinProg", "Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0, 0, 600, 460, NULL, NULL,
                          hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
    {
        DispatchMessage(&Message);
    }
}
```



```

}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_PAINT:
            HDC hdc;
            PAINTSTRUCT ps;
            hdc = BeginPaint (hWnd, &ps);

            HPEN hPen, hPenVorher;
            HGDI OBJ hGDI;
            hPen = CreatePen (PS_SOLID, 2, RGB(255,0,0));
            hGDI = SelectObject (hdc, HGDI OBJ (hPen));
            hPenVorher = HPEN (hGDI);
            HBRUSH hBrush, hBrushVorher;
            hBrush = CreateSolidBrush ( RGB(0,0,255));
            hGDI = SelectObject (hdc, hBrush);
            hBrushVorher = HBRUSH (hGDI);

            Rectangle (hdc, 10, 10, 200, 200);

            Ellipse (hdc, 11, 11, 199, 199);

            SelectObject (hdc, hPenVorher);
            DeleteObject (hPen);
            SelectObject (hdc, hBrushVorher);
            DeleteObject (hBrushVorher);
            EndPaint (hWnd, &ps);

            return 0;
        case WM_DESTROY:

```

```

    PostQuitMessage(0);
    return 0;
default:
    return DefWindowProc (hWnd, uiMessage,
                          wParam, lParam);
}
}

```

### 17.2.3 Textausgabe

Es wird ein Text ausgegeben, dem eine Farbe und eine Hintergrundfarbe zugewiesen wird. Der Text wird ausgerichtet und er enthält eine Schriftart.

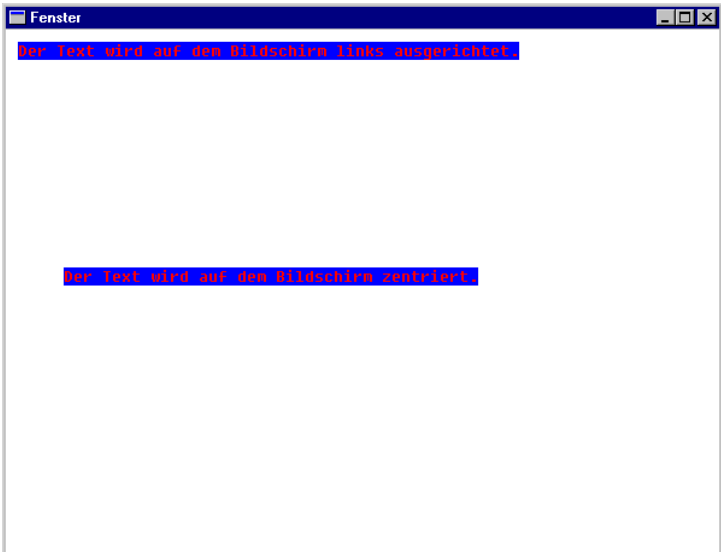


Bild 17.3: Die Anwendung nach dem Start

#### Quelltext

```

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

```

```

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
    hWindow = CreateWindow("WinProg", "Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0, 0, 600, 460, NULL, NULL,
                          hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
    {
        DispatchMessage(&Message);
    }
    return (Message.wParam);
}
LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)

```

```
{
    switch(uiMessage)
    {
        case WM_PAINT:
            HDC hdc;
            PAINTSTRUCT ps;
            hdc = BeginPaint (hWnd, &ps);

            SetTextColor (hdc, RGB(255,0,0));
            SetBkColor (hdc, RGB(0,0,255));
            SetTextAlign (hdc, TA_LEFT);

            SelectObject (hdc,
                GetStockObject
                (SYSTEM_FIXED_FONT));
            char *Text;
            Text = new char[80];

            lstrcpy (Text,
                "Der Text wird auf dem Bildschirm links
                ausgerichtet.");

            TextOut (hdc, 10,10, Text, lstrlen(Text));

            lstrcpy (Text, "Der Text wird auf dem Bildschirm
                zentriert.");

            RECT rect;
            rect.left = 40;
            rect.top = 10;
            rect.right = 400;
            rect.bottom = 400;

            DrawText (hdc, Text, -1, &rect,
                DT_CENTER | DT_VCENTER |
                DT_SINGLELINE);

            delete [] Text;
    }
}
```

```

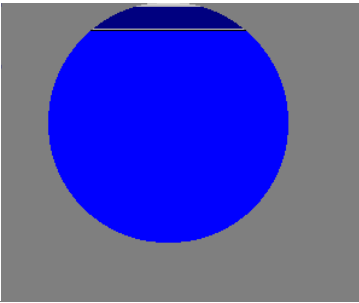
        EndPaint (hWnd, &ps);

        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc (hWnd, uiMessage,
                               wParam, lParam);
    }
}

```

#### 17.2.4 Regionen benutzen

Es wird eine Fensterregion erstellt und dem Fenster zugewiesen. Alle Zeichenfunktionen werden nur noch in diesem Bereich ausgeführt. Dabei ist es egal, ob die Zeichenfunktionen sich auf den Gerätekontext der *Non-Client-Area* beziehen oder auf den Clientbereich des Fensters, denn beide sind mit dem Fenster verbunden. Und so wird jede Zeichenfunktion nur in den Bereich zeichnen, in dem es ihr erlaubt ist. Auf dem Bildschirm wird ein rundes Fenster dargestellt.



*Bild 17.4: Die Anwendung nach dem Start  
(Der graue Teil ist der Hintergrund)*

## Quelltext

```
#include <windows.h>

int Status1;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    WndClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
    hWindow = CreateWindow("WinProg","Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0,0,600,460,NULL,NULL,
                          hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    HRGN hRgn;
    hRgn = CreateEllipticRgn (40, 0, 240, 200);
```

```

SetWindowRgn (hWindow, hRgn, TRUE);

SetTimer (hWindow, 99, 10, 0);

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
    DispatchMessage(&Message);
}

return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_PAINT:
            HDC hdc;
            PAINTSTRUCT ps;
            hdc = BeginPaint (hWnd, &ps);

            SelectObject (hdc,
                          CreateSolidBrush (RGB(0,0,255)));

            Rectangle (hdc,0,0,400,400);

            EndPaint (hWnd, &ps);
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default:
            return DefWindowProc (hWnd, uiMessage,
                                   wParam, lParam);
    }
}

```





# 18 Win32-API: Dateiverwaltung

## 18.1 Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die Dateiverwaltung

Die Win32-API stellt eine Menge von Funktionen für die Dateiverwaltung bereit. Dabei ist sie unabhängig vom jeweiligen Dateisystem, das unter Windows genutzt wird.

Es gibt hauptsächlich drei Dateisysteme, das FAT-Dateisystem, das Protected-FAT-Dateisystem und das NTFS-Dateisystem.

Das FAT-Dateisystem ist das älteste und am häufigsten genutzte Dateisystem. Es kann aber auch genauso gut für Festplatten genutzt werden. Dieses Dateisystem wird von DOS, Windows und OS/2 unterstützt. Die Anzahl der Zeichen für einen Dateinamen beträgt 8 Zeichen.

Das Protected-FAT-Dateisystem ist für Diskettenlaufwerke und Festplatten einsetzbar. Es ist kompatibel mit dem FAT-Dateisystem. Die Länge von Dateinamen kann beim Protected FAT-Dateisystem mit dem NULL-Zeichen 255 Zeichen betragen. Verzeichnispfade können mit dem NULL-Zeichen 260 Zeichen haben

Das NTFS-Dateisystem ist für Windows NT gedacht. Die Zeichenlänge von Dateinamen beträgt ebenfalls 255 Zeichen. Außerdem hat das Dateisystem viele Neuerungen, die es erlauben, den Dateien verschiedene Eigenschaften zuzuweisen.

Die Dateifunktionen arbeiten auf die übliche Art und Weise. Falls eine Datei angesprochen werden soll, wird dafür ein Objekt erstellt. Dieses Dateiojekt kann z. B. einem anderen Programm mitteilen, dass es keinen Zugriff auf die Datei mehr hat, weil schon ein Objekt für diese Datei besteht. Alle Dateifunktionen benötigen diese Dateiojekte, um Aktionen mit den Dateien auszuführen. Das heißt für den Zugriff auf eine Datei wird ein Dateiojekt erstellt, über das der gesamte Zugriff auf diese Datei geregelt wird.

Funktionen, Strukturen, Nachrichten und Objekte	Bedeutung
CreateFile	Die Funktion CreateFile erstellt oder öffnet eine Datei. Auf diese Datei wird ein Datei-Handle zurückgeliefert, also auch ein Dateiobjekt erstellt.
CloseHandle	Die Funktion CloseHandle löscht ein Dateiobjekt.
ReadFile	Die Funktion ReadFile liest Daten aus einer Datei unter Angabe des Dateiobjekts.
WriteFile	Die Funktion WriteFile schreibt Daten in eine Datei unter Angabe des Dateiobjekts.
CopyFile	Die Funktion CopyFile kopiert eine Datei.
DeleteFile	Die Funktion DeleteFile löscht eine Datei.
MoveFile	Die Funktion MoveFile benennt eine Datei oder ein Verzeichnis um.

*Tabelle 18.1: Funktionen, Strukturen, Nachrichten und Objekte der Win32-API für die Dateiverwaltung*

### 18.1.1 CreateFile

Die Funktion CreateFile erstellt ein Objekt für eine Datei, das dafür sorgt, dass kein anderes für diese Datei erstellt werden kann. Es bewirkt, dass die anderen Funktionen, wie ReadFile und WriteFile Zugriff auf die Datei haben.

```
HANDLE CreateFile( LPCTSTR lpFileName,
                  DWORD dwDesiredAccess,
                  DWORD dwShareMode,
                  LPSECURITY_ATTRIBUTES
                  lpSecurityAttributes,
                  DWORD dwCreationDisposition,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile);
```

- E: lpFileName ist der Dateiname. Von dieser Datei wird ein Dateiobjekt erstellt.

- **E:** `dwDesiredAccess` legt fest, wie die Datei verwendet werden darf. Eine Kombination von den folgenden Konstanten wird dafür angegeben:

`GENERIC_WRITE` bestimmt, dass in die Datei geschrieben werden darf.

`GENERIC_READ` bewirkt, dass aus der Datei gelesen werden darf.  
`o` besagt, dass Informationen über ein Gerät ermittelt werden können, ohne das Gerät zu kontaktieren.

- **E:** `dwShareMode` legt fest, ob auf ein Dateiobjekt mehrmals ein Handle wiedergegeben werden kann. Diese Eigenschaft wird durch eine Kombination der folgenden Konstanten festgelegt:

`FILE_SHARE_READ` besagt, dass durch einen Lesezugriff mehrmals ein Handle auf das Objekt wiedergegeben werden kann.

`FILE_SHARE_WRITE` erreicht, dass durch einen Schreibzugriff mehrmals ein Handle auf das Objekt wiedergegeben werden kann.

- **E:** `lpSecurityAttributes` bestimmt, welche Sicherheitseigenschaften das zurückgelieferte Objekt hat. Dies wird durch eine Struktur vom Typ `SECURITY_ATTRIBUTES` angegeben.

- **E:** `dwCreationDisposition` gibt an, welche Aktion ausgeführt werden soll, falls die Datei existiert bzw. nicht existiert. Dieser Parameter setzt sich aus einem der folgenden Werte zusammen:

`CREATE_NEW` bestimmt, dass die Datei neu erstellt wird. Die Funktion liefert einen Fehler zurück, falls die Datei schon existiert.

`CREATE_ALWAYS` bewirkt, dass die Datei immer wieder neu erstellt wird.

`OPEN_EXISTING` legt fest, dass die Datei nur geöffnet wird, wenn sie existiert.

`OPEN_ALWAYS` erreicht, dass die Datei immer geöffnet wird.

`TRUNCATE_EXIST` setzt fest, dass die Datei geöffnet und dann auf `o` Bytes reduziert wird. Falls die Datei nicht existiert, liefert die Funktion einen Fehler zurück.

- **E/A:** `dwFlagsAndAttributes` bestimmt die Dateieigenschaften. Alle Kombinationen der folgenden Konstanten sind möglich:

`FILE_ATTRIBUTE_ARCHIVE` besagt, dass die Datei von einem Backup stammt oder dass sie gelöscht werden kann.

`FILE_ATTRIBUTE_HIDDEN` legt fest, dass die Datei versteckt ist.

`FILE_ATTRIBUTE_NORMAL` gibt an, dass die Datei keine anderen Eigenschaften hat.

`FILE_ATTRIBUTE_OFFLINE` bestimmt, dass die Daten dieser Datei nicht sofort verfügbar sind, da sie nur online vorhanden sind.

`FILE_ATTRIBUTE_READ_ONLY` erklärt, dass die Datei nur gelesen werden kann.

`FILE_ATTRIBUTE_SYSTEM` besagt, dass die Datei ein Teil des Betriebssystems ist oder von dem Betriebssystem genutzt wird.

`FILE_ATTRIBUTE_TEMPORARY` gibt an, dass die Datei eine Auslagerungsdatei ist.

Außerdem sind auch noch folgende, zusätzliche Kombinationen möglich:

`FILE_FLAG_WRITE_THROUGH` bestimmt, dass direkt auf die Festplatte geschrieben wird. Natürlich wird vorher in alle *Cache-Speicher* geschrieben.

`FILE_FLAG_OVERLAPPED` bewirkt, dass Operationen eine bestimmte Zeit für einen Zugriff auf die Datei über ein Dateiobjekt benötigen.

`FILE_FLAG_NO_BUFFERING` gibt an, dass das System die Datei ohne Zwischenspeicher und ohne den *Cache-Speicher* zu benutzen öffnet.

`FILE_FLAG_RANDOM_ACCESS` besagt, dass der Zugriff auf die Datei zufällig erfolgt.

`FILE_FLAG_SEQUENTIAL_SCAN` bestimmt, dass die Datei in einem Stück gelesen wird.

`FILE_FLAG_DELETE_ON_CLOSE` erreicht, dass die Datei gelöscht wird, nachdem alle Handles auf die Datei geschlossen worden sind.

FILE\_FLAG\_BACKUP\_SEMANTIC besagt, dass die Datei für einen Backup geöffnet oder erstellt wurde.

FILE\_POSIX\_SEMATICS definiert, dass die Datei nach *POSIX-Regeln* verwaltet wird.

FILE\_FLAG\_OPEN\_REPARSE\_POINT bestimmt das Verhalten des *reparse Points*.

FILE\_FLAG\_OPEN\_NO\_RECALL legt fest, wie Daten der Datei gespeichert werden.

Auch eine Kombination der folgenden Konstanten kann angegeben werden:

SECURITY\_ANONYMOUS bestimmt, dass der Client anonym ist.

SECURITY\_IDENTIFICATION besagt, dass der Client sich identifizieren muss.

SECURITY\_IMPERSONATION setzt fest, dass der Client sich auf dem *Impersonate-Level* befinden muss.

SECURITY\_DELEGATION gibt an, dass der Client sich auf dem *Delegations.-Level* befinden muss.

SECURITY\_CONTEXT\_TRACKING bestimmt, dass die Sicherheitsverfolgung dynamisch ist, andernfalls ist sie statisch.

SECURITY\_EFFECTIVE\_ONLY bestimmt, dass nur die eingeschalteten Clientaspekte für den Server sichtbar sind.

- E: Definiert einen Handle für eine *Template-Datei*.
- A/R: Falls die Funktion ohne Fehler ausgeführt werden kann, liefert die Funktion den Handle auf ein Dateiojekt zurück. Falls die Datei überschrieben wird, liefert die Funktion `GetLastError` den Wert der Konstanten `ERROR_ALREADY_EXISTS` zurück.
- N: Falls während der Ausführung der Funktion ein Fehler auftritt, liefert die Funktion den Wert von der Konstanten `INVALID_FILE_HANDLE` zurück. Weitere Informationen über den Fehler liefert die Funktion `GetLastError`.

Unterstützung:

- Windows NT 3.1 und später

- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINBASE.H

Library-Datei: KERNEL32.LIB

Unterstützung für ANSI und UNICODE unter Windows NT

### 18.1.2 CloseHandle

Die Funktion `CloseHandle` kann ein offenes Dateiojekt schließen. Das heißt die Funktion sorgt dafür, dass das Dateiojekt einen neuen Handle herausgeben kann.

```
BOOL CloseHandle( HANDLE hObject);
```

- E: `hObject` ist der Handle auf das Dateiojekt, das geschlossen werden soll.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich NULL, falls bei der Ausführung keine Fehler aufgetreten sind.
- N: Falls bei der Ausführung der Funktion jedoch Fehler auftreten, liefert die Funktion den Wert NULL zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINBASE.H

Library-Datei: KERNEL32.LIB

### 18.1.3 ReadFile

Die Funktion `ReadFile` liest unter Angabe eines Dateiobjekts Daten aus einer Datei.

```
BOOL ReadFile( HANDLE hFile, LPVOID lpBuffer,  
              DWORD nNumberOfBytesToRead,  
              LPDWORD lpNumberOfBytesRead,  
              LPOVERLAPPED lpOverlapped);
```

- E: `hFile` ist der Handle auf das Dateiobjekt, aus dem gelesen werden soll. Das Dateiobjekt muss mit der Konstanten `GENERIC_READ` erstellt worden sein.
- E: `lpBuffer` ist ein Zeiger auf den Speicher, in den die Datei eingelesen werden soll.
- E: `nNumberOfBytesToRead` gibt die Anzahl der Bytes an, die aus der Datei zu lesen sind.
- E: `lpNumberOfBytesRead` gibt die Anzahl der Bytes an, die aus der Datei gelesen wurden.
- E: `lpOverlapped` ist ein Zeiger auf eine Struktur vom Typ `OVERLAPPED`. In dieser Struktur kann z. B. ein Offset in die Datei eingegeben werden. Das Dateiobjekt muss mit der Konstante `FILE_FLAG_OVERLAPPED` erstellt worden sein.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, falls bei der Ausführung keine Fehler aufgetreten sind.
- N: Falls bei der Ausführung der Funktion jedoch Fehler auftreten, liefert die Funktion den Wert `NULL` zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINBASE.H`

Library-Datei: `KERNEL32.LIB`

### 18.1.4 WriteFile

Die Funktion `WriteFile` schreibt Daten in eine Datei, die mit einem Dateiobjekt angegeben worden ist.

```
BOOL WriteFile( HANDLE hFile, LPCVOID lpBuffer,
                DWORD nNumberOfBytesToWrite,
                LPDWORD lpNumberOfBytesWritten,
                LPOVERLAPPED lpOverlapped);
```

- E: `hFile` ist ein Handle auf ein Dateiojekt. Die Datei, die von diesem Objekt repräsentiert wird, ist die Datei, aus der die Daten gelesen werden. Das Dateiojekt muss mit der Konstanten `GENERIC_WRITE` erstellt worden sein.
- E: `lpBuffer` ist ein Zeiger auf einen Buffer, aus dem die Daten geschrieben werden.
- E: `nNumberOfBytesToWrite` ist die Anzahl der Bytes, die geschrieben werden sollen.
- E: `lpNumberOfBytesWritten` ist die Anzahl der Bytes, die geschrieben wurden.
- E: `lpOverlapped` ist ein Zeiger auf eine Struktur vom Typ `OVERLAPPED`. Mit dieser Struktur ist es möglich, einen Offset in die Datei einzugeben. Das Dateiojekt muss mit der Konstanten `FILE_FLAG_OVERLAPPED` erstellt worden sein.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich `NULL`, falls bei der Ausführung keine Fehler aufgetreten sind.
- N: Falls bei der Ausführung der Funktion jedoch Fehler auftreten, liefert die Funktion den Wert `NULL` zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: `WINBASE.H`

- Library-Datei: `KERNEL32.LIB`

### 18.1.5 CopyFile

Die Funktion `CopyFile` kopiert eine Datei, indem sie eine andere Datei erstellt und dort denselben Inhalt ablegt.



```
BOOL CopyFile( LPCTSTR lpExistingFileName,
               LPCTSTR lpNewFileName, BOOL bFailIfExists);
```

- E: `lpExistingFileName` ist der Name der Datei, die kopiert werden soll.
- E: `lpNewFileName` ist der Name der Datei, die für den Kopiervorgang neu erstellt wird.
- E: `bFailIfExists` bestimmt, wie sich die Funktion verhalten soll, falls die Datei `lpNewFileName` schon existiert.

TRUE legt fest, dass ein Fehler ausgegeben wird, falls die Datei schon existiert.

FALSE bestimmt, dass die Datei überschrieben wird.

- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich NULL, falls bei der Ausführung keine Fehler aufgetreten sind.
- N: Falls bei der Ausführung der Funktion jedoch Fehler auftreten, liefert die Funktion den Wert NULL zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINBASE.H

Library-Datei: KERNEL32.LIB

Unterstützung für ANSI und UNICODE unter Windows NT

### 18.1.6 DeleteFile

Die Funktion `DeleteFile` löscht eine Datei.

```
BOOL DeleteFile( LPCTSTR lpFileName);
```

- E: `lpFileName` ist der Name der Datei, die gelöscht werden soll.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich NULL, falls bei der Ausführung keine Fehler aufgetreten sind.
- N: Falls bei der Ausführung der Funktion jedoch Fehler auftreten, liefert die Funktion den Wert NULL zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINBASE.H

Library-Datei: KERNEL32.LIB

- Unterstützung für ANSI und UNICODE unter Windows NT

### 18.1.7 MoveFile

Die Funktion `MoveFile` benennt eine Datei oder ein Verzeichnis um.

```
BOOL MoveFile( LPCTSTR lpExistingFileName,
               LPCTSTR lpNewFileName);
```

- E: `lpExistingFileName` ist der Name der Datei oder eines Verzeichnisses. Die angegebene Datei oder das angegebene Verzeichnis wird umbenannt.
- E: `lpNewFileName` ist der Name, den die Datei oder das Verzeichnis erhält.
- A/R: Als Rückgabewert liefert die Funktion einen Wert ungleich NULL, falls bei der Ausführung keine Fehler aufgetreten sind.
- N: Falls bei der Ausführung der Funktion jedoch Fehler auftreten, liefert die Funktion den Wert NULL zurück.

Unterstützung:

- Windows NT 3.1 und später
- Windows 95 und später
- Windows CE 1.0 und später

Header-Datei: WINBASE.H

Library-Datei: KERNEL32.LIB

- Unterstützung für ANSI und UNICODE unter Windows NT

## 18.2 Beispiele

### 18.2.1 Eine einfache Datei erstellen und sie mit Daten füllen

Dieses Beispiel speichert einen `int`-Wert in einer Datei ab.

#### Quelltext

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    HANDLE hFile;
    int Daten;
    Daten = 1;
    DWORD Readd;
    hFile = CreateFile ("c:\\int.dat", GENERIC_WRITE, 0,
                      NULL, CREATE_ALWAYS, NULL, NULL);
    WriteFile (hFile,&Daten,sizeof(Daten),&Readd,NULL);
    CloseHandle (hFile);

    return 0;
}
```

### 18.2.2 Eine einfache Datei öffnen und Daten aus ihr lesen

Dieses Beispiel öffnet die Datei `INT.DAT` und liest aus ihr einen `int`-Wert, der dann auf dem Bildschirm ausgegeben wird. Für diesen Zweck wird kein Gerätekontext für ein Fenster, sondern für den gesamten Bildschirm ermittelt.

#### Quelltext

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
```

```
int nCmdShow )
{
    HANDLE hFile;
    int Daten;
    DWORD Readd;
    hFile = CreateFile ("c:\\int.dat", GENERIC_READ, 0,
                      NULL, OPEN_EXISTING, NULL, NULL);
    ReadFile (hFile,&Daten,sizeof(Daten),&Readd,NULL);
    CloseHandle (hFile);

    HDC hdc;
    hdc = GetDC (NULL);

    char *Text;
    Text = new char[20];

    itoa (Daten, Text, 10);

    TextOut (hdc, 10, 10, Text, lstrlen (Text));

    delete [] Text;

    ReleaseDC (NULL, hdc);

    return 0;
}
```

# 19

## Vordefinierte Fensterklassen

### 19.1 Allgemeines

Unter Windows gibt es mehrere vordefinierte Fensterklassen, die hauptsächlich für Steuerelemente genutzt werden. Alle Fensterklassen haben auch ihre eigene Fensterfunktion. Die wichtigste Klasse heißt `BUTTON`, mit der, wie der Name sagt, z. B. Buttons gesetzt werden. Die Fensterklassen können von Windowsversion zu Windowsversion variieren. Unter den Standard-Windowsversionen ändern sich die Proportionen aber nicht, damit eine Abwärtskompatibilität gesichert wird.

Alle Fenster, die durch eine vordefinierte Fensterklasse erstellt werden, sind so konzipiert, dass sie bei einem Ereignis eine Nachricht an das übergeordnete Fenster liefern. Diese Nachricht heißt bei allen Steuerelementen meistens `WM_COMMAND`.

Natürlich kann man auch Nachrichten an die Steuerelemente senden, durch die ein simulierter Umgang mit Steuerelementen möglich wird. Dieser ist auch durch das Senden von Systemnachrichten oder durch das Senden einer Nachricht `WM_COMMAND` an das übergeordnete Fenster möglich.

Um die Farbe von bestimmten Objekten zu ändern, die durch vordefinierte Fensterklassen erzeugt wurden, verwendet man die Nachricht `WM_CTLCOLOR`.

Fensterklasse	Bedeutung
<code>BUTTON</code>	Das Fenster kann ein Push-Button, eine CheckBox, ein Radio-Button, ein Owner-Drawn-Button oder eine GroupBox sein.
<code>EDIT</code>	Das Fenster ist ein Edit-Feld.

Fensterklasse	Bedeutung
LISTBOX	Das Fenster ist eine ListBox.
STATIC	Das Fenster ist ein Textfeld.

Tabelle 19.1: Wichtige Fensterklassen

### 19.1.1 BUTTON

#### Allgemeines

Es gibt fünf verschiedene Arten von Buttons, nämlich

1. Push-Buttons
2. CheckBoxes
3. Radio-Buttons
4. Owner-drawn-Buttons
5. GroupBoxes

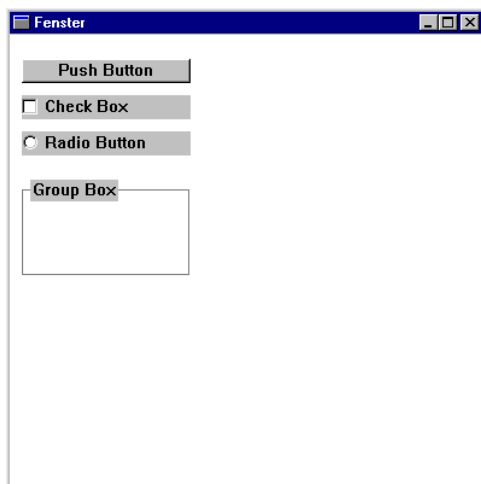


Bild 19.1: Die verschiedenen Buttons

### *Push-Button*

Ein Push-Button ist ein rechteckiger Kasten mit Beschriftung oder Bild. Dieser Kasten kann angeklickt werden, woraufhin eine Aktion ausgelöst wird.

### *CheckBox*

Eine CheckBox ist ein rechteckiges Feld mit einer Beschriftung oder einem Bild, die neben der CheckBox angebracht werden. Die CheckBox kann markiert werden und sollte dann eingesetzt werden, wenn ein oder mehrere Objekte ausgewählt werden können.

### *Radio-Button*

Ein Radio-Button ist ein rundes Feld mit einer Beschriftung oder einem Bitmap, das seitwärts angebracht wurde. Wenn ein Radio-Button markiert wird, sollten sich alle anderen zugehörigen Radio-Buttons auf „nicht markiert“ zurückstellen.

### *Owner-drawn-Buttons*

Ein Owner-drawn-Button ist ein Feld, das die Möglichkeit bietet, einen Button selber zu zeichnen.

### *GroupBox*

Eine GroupBox ist ein Rechteck, das eine Gruppe von Kontrollelementen einschließt. Dies dient der optischen Darstellung. In der oberen linken Ecke befindet sich ein Text.

## **Styles**

`BS_PUSHBUTTON` bestimmt, dass das erstellte Fensterobjekt der `BUTTON`-Klasse wie ein Push-Button aussieht und sich auch so verhält.

`BS_DEFPUSHBUTTON` bewirkt, dass das erstellte Fensterobjekt der `BUTTON`-Klasse wie ein Push-Button aussieht und sich so verhält, als wäre er der Standard-Push-Button.

`BS_CHECKBOX` gibt an, dass das erstellte Fensterobjekt der `BUTTON`-Klasse wie eine Check Box aussieht.

**BS\_AUTOCHECKBOX** bestimmt, dass das erstellte Fensterobjekt der **BUTTON**-Klasse wie eine **CheckBox** aussieht und alle anderen **CheckBox**-Boxen derselben Gruppe nicht aktiviert, falls die **CheckBox** aktiviert wird.

**BS\_RADIOBUTTON** legt fest, dass das erstellte Fensterobjekt der **BUTTON**-Klasse wie ein **Radio-Button** aussieht.

**BS\_AUTORADIOBUTTON** bestimmt, dass das erstellte Fensterobjekt der **BUTTON**-Klasse wie ein **Radio-Button** aussieht und alle anderen **Radio-Buttons** nicht markiert, wenn der jeweilige **Radio-Button** markiert wird.

**BS\_LEFTTEXT** bewirkt, dass bei **RadioBox** und **CheckBox** der Text auf der linken Seite steht.

**BS\_OWNERDRAWN** setzt fest, dass das erstellte Fenster sich wie ein **Button** verhält, wobei der **Button** gezeichnet werden muss.; dazu wird die Nachricht **WM\_DRAWITEM** an das **Parent-Fenster** gesendet.

**BS\_GROUPBOX** bewirkt, dass das erstellte Fenster sich wie ein **Gruppenfenster** verhält. In der oberen linken Ecke kann Text stehen.

**BS\_3STATE** besagt, dass das erstellte Fenster sich wie eine **CheckBox** verhält. Die **CheckBox** kann aber drei Zustände annehmen.

**BS\_AUTO3STATE** bestimmt, dass das erstellte Fenster sich wie eine **CheckBox** mit drei Zuständen verhält, wobei alle anderen nicht markiert werden, wenn die **CheckBox** markiert wird.

**BS\_USERBUTTON** bewirkt dasselbe wie **BS\_OWNERDRAWN**. Dies ist so, weil **BS\_USERBUTTON** nur noch bei **16-Bit-Programmen** wichtig ist.

**BS\_BITMAP** gibt an, dass der **Button** ein **Bitmap** zeigt.

**BS\_BOTTOM** setzt fest, dass der Text am **Boden** ausgerichtet wird.

**BS\_CENTER** definiert, dass der Text **zentriert** ausgerichtet wird.

**BS\_RIGHT** bewirkt, dass der Text **rechts** ausgerichtet wird.

**BS\_ICON** erreicht, dass der **Button** ein **Icon** zeigt.

**BS\_FLAT** gibt an, dass der **Button** **zweidimensional** angezeigt wird.



BS\_MULTILINE bestimmt, dass der Buttontext aus mehreren Zeilen besteht, falls er zu groß ist.

BS\_NOTIFY besagt, dass der Button BM\_SETFOCUS, BM\_KILLFOCUS und BM\_DBLCLICK an das übergeordnete Fenster sendet. Die Nachricht BN\_CLICKED wird immer gesendet.

BS\_PUSHLIKE gibt an, dass andere Buttons so aussehen, als wären sie ein Push-Button. Zu diesen Buttons zählen Check-Buttons, Radio-Buttons und Buttons mit drei Zuständen.

BS\_RIGHTBUTTON bestimmt dasselbe wie BS\_LEFTTEXT.

BS\_TEXT bestimmt, dass der Button Text zeigt.

BS\_TOP bestimmt, dass der Text oben ausgerichtet wird.

BS\_VCENTER bestimmt, dass der Text in der vertikalen Mitte des Buttonrechtecks steht.

### **Nachrichten vom Parent-Fenster an den Button**

#### *BM\_CLICK*

Die Nachricht BM\_CLICK wird an den Button gesendet. Dadurch werden WM\_LBUTTONDOWN und WM\_LBUTTONUP ausgelöst und es wird eine Nachricht BN\_CLICKED an das Parent-Fenster zurückgesandt. Durch diese Nachricht wird also ein Button-Click simuliert. Ansonsten hat diese Nachricht keine Bedeutung, sie ist nur zur Simulation gedacht.

BM\_CLICK

wParam = 0;

lParam = 0;

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## BM\_GETCHECK

Die Nachricht BM\_GETCHECK wird an das Buttonfenster gesendet und liefert zurück, ob der Button markiert oder nicht markiert ist. Dies geschieht nur bei Radio-Buttons und CheckBoxen.

```
BM_GETCHECK
```

```
wParam = 0;
```

```
lParam = 0;
```

- A/R: Als Return liefert die Nachricht die folgenden Konstanten zurück.

BST\_CHECKED bestimmt, dass der Button markiert ist.

BST\_UNCHECKED definiert, dass der Button nicht markiert ist.

BST\_INTERMEDIATE bestimmt, dass der Button grau ist.

Unterstützung: WinNT 3.1 und später/ Win95 und später/ WinCE 1.0 und später

Header-Datei: WINUSER.H

## BM\_GETIMAGE

Die Nachricht BM\_GETIMAGE wird gesendet, um einen Handle auf das Bitmap oder Icon zu erhalten, das mit dem Button assoziiert wurde.

```
BM_GETIMAGE
```

```
wParam = (WPARAM) fImageType;
```

```
lParam = 0;
```

- E: fImageType gibt an, ob ein Icon oder Bitmap zurückgegeben werden soll.

IMAGE\_BITMAP sagt aus, dass ein Handle auf ein Bitmap zurückgegeben werden soll.

IMAGE\_ICON bestimmt, dass ein Handle auf ein Icon zurückgegeben werden soll.

- A/R: Falls ein Bitmap oder Icon gefunden wurde, wird der Handle darauf zurückgeliefert.

- N: Falls kein Bitmap oder Icon vorhanden ist, wird der Wert NULL zurückgeliefert.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE nicht unterstützt

Header-Datei: WINUSER.H

### *BM\_GETSTATE*

Die Nachricht *BM\_GETSTATE* wird gesendet, um herauszufinden, wie der Status eines Buttons oder einer CheckBox ist.

*BM\_GETSTATE*

wParam = 0;

lParam = 0;

- A/R: Zurückgeliefert wird ein Wert, der den Status des Buttons angibt. Mit Hilfe einer Bit-Maske wird der Status des Button ermittelt.

0x0003

*BST\_CHECKED* bestimmt, dass der Button markiert ist.

*BST\_UNCHECKED* bestimmt, dass der Button nicht markiert ist.

*BST\_INTERMEDIATE* gibt an, dass der Button grau ist.

*BST\_PUSHED* setzt fest, dass der Button gerade gedrückt wird.

*BST\_FOCUS* legt fest, dass der Button den Focus hat, falls der Wert ungleich NULL ist.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## BM\_SETCHECK

Die Nachricht BM\_SETCHECK wird benutzt, um die Markierung einer CheckBox oder eines Radio-Button zu setzen.

BM\_SETCHECK

wParam = (WPARAM) fCheck;

lParam = 0;

- E: Als Eingabe wird der Wert einer Konstanten erwartet, der angibt, welchen Zustand die CheckBox oder der Radio-Button haben soll.

BST\_CHECKED bestimmt, dass der Button markiert ist.

BST\_UNCHECKED gibt an, dass der Button nicht markiert ist.

BST\_INTERMEDIATE setzt fest, dass der Button grau ist.

- A/R: Als *return* wird immer der Wert NULL zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## BM\_SETIMAGE

Die Nachricht BM\_SETIMAGE wird gesendet, um einem Button ein Bitmap oder ein Icon zuzuordnen.

BM\_SETIMAGE

wParam = (WPARAM) fImageType;

lParam = (LPARAM) (HANDLE) hImage;

- E: wParam bestimmt, ob ein Bitmap oder Icon zugewiesen werden soll.

IMAGE\_BITMAP gibt an, dass ein Bitmap zugewiesen wird.

IMAGE\_ICON definiert, dass ein Icon zugewiesen wird.

- E: lParam ist der Handle auf das Bitmap oder Icon.

- Als `return` wird ein Handle auf das zuvor eingestellte Bitmap oder Icon zurückgeliefert.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE nicht unterstützt

Header-Datei: `WINUSER.H`

### *BM\_SETSTATE*

Die Nachricht `BM_SETSTATE` bestimmt, ob der Button so aussehen soll, als ob er gedrückt, oder als ob er nicht gedrückt wurde.

`BM_SETSTATE`

`wParam = (WPARAM) fState;`

`lParam = 0;`

- E: `wParam` bestimmt das Aussehen des Buttons.

`TRUE` besagt, dass der Button so aussieht, als wäre er gedrückt.

`FALSE` gibt an, dass der Button so aussieht, als wäre er nicht gedrückt.

- A/R: Diese Nachricht gibt immer den Wert `NULL` zurück.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *BM\_SETSTYLE*

Die Nachricht `BM_SETSTYLE` ändert die Eigenschaften eines Buttons. Die alten Styles, die durch die Funktion `CreateWindow` angegeben wurden, werden durch diese Styles ersetzt.

`BM_SETSTYLE`

`wParam = (WPARAM) LOWORD(dwStyle);`

`lParam = MAKELPARAM(fRedraw, 0);`

- E: `wParam` bestimmt die neuen Eigenschaften des Buttons. Diese Eigenschaften können eine Kombination aller Buttoneigenschaften sein.
- E: `lParam` gibt an, ob der Button neu gezeichnet werden soll.  
TRUE definiert, dass der Button neu gezeichnet werden soll.  
FALSE bestimmt, dass der Button nicht neu gezeichnet werden soll.
- A/R: Als `return` liefert die Nachricht immer NULL zurück.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### Nachrichten vom Button an das Parent-Fenster

#### WM\_COMMAND

Die Nachricht WM\_COMMAND wird dazu benutzt, das übergeordnete Fenster über Ereignissen zu informieren. Sie wird nicht nur von Steuerelementen, sondern auch von Menüs oder von besonderen Tasten verwendet.

WM\_COMMAND

```
wNotifyCode = HIWORD(wParam);
```

```
wID = LOWORD(wParam);
```

```
hwndCtl = (HWND) lParam;
```

- A: `wNotifyCode` definiert die Nachricht genauer. Hier wird die *notification* angegeben.
- A: `wID` bestimmt die ID des Steuerelements. Diese wurde in der Funktion `CreateWindow` durch `hMenu` angegeben.
- A: `hwndCtl` gibt an, von welchem Steuerelement die Nachricht kam.

Unterstützung:

- WinNT 3.1 und später

- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *WM\_CTLCOLORBTN*

Die Nachricht *WM\_CTLCOLORBTN* wird gesendet, damit das Parent-Fenster die Farbe des Buttons ändern kann.

*WM\_CTLCOLORBTN*

*hdcButton* = (HDC) *wParam*;

*hwndButton* = (HWND) *lParam*;

- **A:** *hdcButton* bestimmt den Gerätekontext des Buttons.
- **A:** *hwndButton* definiert das Fensterobjekt des Buttons.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 2.0 und später

Header-Datei: WINUSER.H

### *WM\_DRAWITEM*

Die Nachricht *WM\_DRAWITEM* wird gesendet, wenn bei einem Ownerdrawn-Button etwas neu gezeichnet werden soll.

*WM\_DRAWITEM*

*idCtl* = (UINT) *wParam*;

*lpDrawStruct* = (LPDRAWITEMSTRUCT) *lParam*;

- **A:** *idCtl* ist die ID des Buttons.
- **A:** *lpDrawStruct* ist ein Zeiger auf eine Struktur vom Typ *DRAWITEMSTRUCT*.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## DRAWITEMSTRUCT

Die Struktur DRAWITEMSTRUCT enthält Informationen, die angeben, was neu gezeichnet werden muss.

```
typedef struct tagDRAWITEMSTRUCT
{
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    UINT itemAction;
    UINT itemState;
    HWND hwndItem;
    HDC hDC;
    RECT rcItem;
    DWORD itemData;
} DRAWITEMSTRUCT;
```

- `CtlType` bestimmt, ob ein Button oder ein anderes Steuerelement neu gezeichnet werden soll.

`ODT_BUTTON` gibt an, dass ein Button neu gezeichnet werden soll.

`ODT_COMBOBOX` legt fest, dass eine ComboBox neu gezeichnet werden soll.

`ODT_LISTBOX` bewirkt, dass eine ListBox neu gezeichnet werden soll.

`ODT_LISTVIEW` erreicht, dass eine ListViewBox neu gezeichnet werden soll.

`ODT_MENU` bestimmt, dass ein Menü neu gezeichnet werden soll.

`ODT_STATIC` besagt, dass ein Textfeld neu gezeichnet werden soll.

`ODT_TAB` bestimmt, dass ein Kontrollelement neu gezeichnet werden soll.

- `CtlID` definiert die ID des Steuerelements, das neu gezeichnet werden soll.
- `itemID` bestimmt den Menüpunkt, der neu gezeichnet werden soll.



- `itemAction` gibt an, was neu gezeichnet werden soll.  
ODA\_DRAWENTIRE erklärt, dass alles neu gezeichnet werden soll.  
ODA\_FOCUS entscheidet, dass das Steuerelement den Fokus hat bzw. nicht mehr hat.  
ODA\_SELECT gibt an, ob das Steuerelement ausgewählt oder losgelassen wurde.
- `itemState` bestimmt den Zustand des Steuerelements, nachdem es neu gezeichnet wurde.  
ODS\_CHECKED gibt an, dass das Steuerelement markiert wurde.  
ODS\_COMBOBOXEDIT setzt fest, dass eine ComboBox editiert wurde.  
ODS\_DEFAULT regelt, dass das Steuerelement das Standard-Steuerelement ist.  
ODS\_DISABLED bestimmt, dass das Steuerelement ausgeschaltet wurde.  
ODS\_FOCUS bewirkt, dass das Steuerelement den Fokus hat.  
ODS\_GRAYED setzt fest, dass das Steuerelement grau ist.  
ODS\_SELECTED bestimmt, dass der Menüpunkt ausgewählt wurde.
- `hwndItem` regelt einen Handle auf das Fensterobjekt.
- `hDC` bewirkt einen Handle auf einen Gerätekontext.
- `rcItem` bestimmt das Rechteck des Steuerelements.
- `itemData` beinhaltet eine Zusatzinformation des Steuerelementes.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## Notification-Nachrichten

BN\_CLICKED bestimmt, dass ein Button betätigt wurde.

BN\_DBLCLCK/ BN\_DOUBLECLICKED gibt an, dass ein Button mit einem Doppelklick betätigt wurde.

BN\_DISABLE vermerkt, dass ein Button ausgeschaltet wurde.

BN\_HILITE/ BN\_PUSHED sagt, dass ein Button ausgewählt wurde.

BN\_UNHILITE/ BN\_UNPUSHED gibt an, dass ein Button wieder losgelassen wurde.

BN\_SETFOCUS besagt, dass ein Button den Eingabefokus erhält.

BN\_KILLFOCUS regelt, dass ein Button den Eingabefokus verliert.

### 19.1.2 EDIT

Ein Edit-Feld ist ein Eingabefeld, das dazu dient, Eingaben vom Benutzer aufzunehmen.

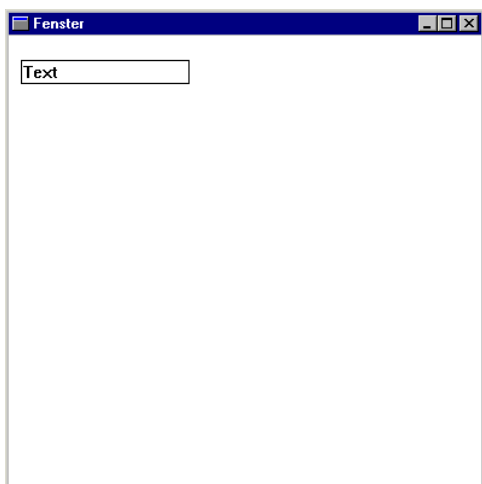



Bild 19.2: Ein Edit-Feld

## styles

ES\_AUTOHSCROLL bestimmt, dass automatisch zehn Zeichen weiter nach rechts gesprungen wird. Durch Betätigen von  erreicht man den Anfang.

ES\_AUTOVSCROLL bewirkt, dass automatisch eine Seite weitergeblättert wird, wenn die Entertaste betätigt wird.

ES\_CENTER gibt an, dass der Text in `singleline` und `multiline` zentriert wird. Die Zentrierung von Text in `singleline` ist nur unter Windows NT 5.0 und Windows 98 möglich.

ES\_LEFT besagt, dass der Text links ausgerichtet wird.

ES\_LOWERCASE bestimmt, dass alle Zeichen Kleinbuchstaben werden, wenn sie in das Edit-Feld eingegeben werden.

ES\_MULTILINE gibt an, dass das Edit-Feld mehrere Zeilen beinhalten kann. Ein Edit-Feld mit ES\_MULTILINE kann Scrollbars enthalten. Normalerweise können Edit-Felder nur eine Textzeile aufnehmen.

ES\_NOHIDSEL sagt, dass die Markierungen in einem Edit-Feld bleibt, auch wenn das Fenster nicht mehr den Eingabe Fokus hat.

ES\_OEMCONVERT regelt, dass der Eingegebene ANSI-Text in den OEM Zeichensatz umgewandelt wird und dann wieder zurückgewandelt wird.

ES\_PASSWORD bestimmt, dass alle eingegebenen Zeichen durch ein anderes Zeichen (asterisk \*) ersetzt werden.

ES\_RIGHT sagt, dass der Text in einem einzeiligen Edit-Feld am rechten Rand ausgerichtet sind.

ES\_UPPERCASE bestimmt, dass alle Zeichen, die in das Edit-Feld eingegeben werden, zu großen Zeichen werden.

ES\_READONLY bewirkt, dass der Benutzer keinen Text in das Edit-Feld eingeben kann und auch keine Veränderungen daran vornehmen kann.

ES\_WANTRETURN gibt an, dass ein *carriage return* in den Text eingefügt wird. Andernfalls wird der Standard-Push-Button aktiviert.

ES\_NUMBER besagt, dass nur Zahlen in das Edit-Feld eingetragen werden können.

## Nachrichten vom Parent-Fenster an das Edit-Feld

Zu den Edit-Feldern wird auch die Verarbeitung verschiedener Standardnachrichten angegeben, weil diese Informationen wichtig sein könnten, z.B. weil diese Nachrichten auch an das Edit-Feld gesendet werden und gewünschte Aktionen auslösen können.

### *EM\_CANUNDO*

Die Nachricht *EM\_CANUNDO* wird gesendet, um Informationen darüber zu bekommen, ob die letzte Aktion im Edit-Feld durch die Nachricht *EM\_UNDO* rückgängig gemacht werden kann.

*EM\_CANUNDO*

wParam = 0;

lParam = 0;

- A/R: Falls *EM\_UNDO* erfolgreich ausgeführt wird, wird **TRUE** zurückgeliefert, andernfalls **FALSE**.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: **WINUSER.H**

### *EM\_UNDO*

Die Nachricht *EM\_UNDO* wird gesendet, um die letzte Aktion im Edit-Feld rückgängig zu machen.

*EM\_UNDO*

wParam = 0;

lParam = 0;

- A/R: Bei einem einzeiligen Edit-Feld wird als `return` immer **TRUE** zurückgeliefert, wenn die Funktion erfolgreich ausgeführt worden ist. Falls die Funktion nicht ausgeführt werden kann, wird der Wert **FALSE** zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später

- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_CHARFROMPOS*

Die Nachricht EM\_CHARFROMPOS wird gesendet, um den Index des Zeichens und die Nummer der Zeile zu erhalten.

EM\_CHARFROMPOS

wParam = 0;

lParam = (LPARAM) (POINTL \*) lpPoint;

- E: lParam bestimmt durch eine POINT-Struktur, die Position, von der die Daten zurückgeliefert werden sollen.
- A/R: Im unteren WORD des Wertes befindet sich der Index des Zeichens. Im oberen WORD befindet sich der Index der Zeile. Alle Nummerierungen beginnen mit 0.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_EMPTYUNDOBUFFER*

Die Nachricht EM\_EMPTYUNDOBUFFER wird gesendet, um den Buffer für die Funktion EM\_UNDO zu entleeren.

EM\_EMPTYUNDOBUFFER

wParam = 0;

lParam = 0;

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## EM\_FMTLINES

Die Nachricht EM\_FMTLINES wird gesendet, damit in mehrzeiligen Edit-Feldern weiche `returns` eingefügt werden können.

EM\_FMTLINES

wParam = (WPARAM) (BOOL) fAddEOL;

lParam = 0;

- E: wParam bestimmt, ob weiche `returns` eingefügt werden dürfen. Durch den Wert TRUE wird dies erlaubt. Der Wert FALSE erlaubt dies nicht.
- A/R: Der `return`-Wert entspricht dem Wert von wParam.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## EM\_GETFIRSTVISIBLELINE

Die Nachricht EM\_GETFIRSTVISIBLELINE wird gesendet, um die Nummer der ersten Linie im Edit-Feld zu ermitteln.

EM\_GETFIRSTVISIBLELINE

wParam = 0;

lParam = 0;

- A/R: Als `return` wird der Index-Wert der Linie zurückgeliefert. Der Index-Wert basiert auf der Zahl 0.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## EM\_GETHANDLE

Die Nachricht EM\_GETHANDLE wird an ein Edit-Feld gesendet, um den Handle des Textbuffers eines mehrzeiligen Edit-Feldes zu ermitteln.

```
EM_GETHANDLE
```

```
wParam = 0;
```

```
lParam = 0;
```

- A/R: Als `return` bekommt man einen Handle auf den Speicher, in dem der Text abgespeichert ist.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 2.0 und später

Header-Datei: WINUSER.H

## EM\_GETLIMITTEXT

Die Nachricht EM\_GETLIMITTEXT wird gesendet, um die maximale Länge des Texts zu bekommen.

```
EM_GETLIMITTEXT
```

```
wParam = 0;
```

```
lParam = 0;
```

- A/R: Als `return` wird die maximale Länge des Texts zurückgeliefert.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## EM\_GETLINE

Die Nachricht EM\_GETLINE wird gesendet, um eine Textzeile aus einem Edit-Feld zu kopieren.

EM\_GETLINE

wParam = (WPARAM) line;

lParam = (LPARAM) (LPCSTR) lpch;

- wParam bestimmt den Index der Linie. Dieser Wert basiert auf NULL. Bei einem einzeiligen Edit-Feld wird dieser Parameter ignoriert.
- lParam bestimmt den Zeiger auf den Textbuffer.
- A/R: Falls die Funktion erfolgreich ausgeführt wurde, wird die Anzahl der Zeichen zurückgeliefert, die kopiert worden sind.  
N: Falls die Anzahl der Linien größer ist, als die Anzahl, die tatsächlich vorhanden ist, wird der Wert NULL zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

*EM\_GETLINECOUNT*

Die Nachricht EM\_GETLINECOUNT wird gesendet, um die Anzahl der Linien in einem mehrzeiligen Edit-Feld herauszubekommen.

EM\_GETLINECOUNT

wParam = 0;

lParam = 0;

- A/R: Falls die Funktion erfolgreich ausgeführt wird, wird die Anzahl der Zeilen des Edit-Feldes zurückgeliefert. Falls kein Text im Edit-Feld enthalten ist, wird der Wert 1 zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H



## *EM\_GETMARGIN*

Die Nachricht *EM\_GETMARGIN* wird gesendet, um die Breite des linken und rechten Randes festzustellen.

*EM\_GETMARGINS*

wParam = 0;

lParam = 0;

- A/R: Im unteren Wort befindet sich die Breite des linken Randes. Im oberen Wort befindet sich die Breite des rechten Randes.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## *EM\_GETMODIFY*

Die Nachricht *EM\_GETMODIFY* wird gesendet, um herauszufinden, ob der Inhalt eines Edit-Feldes verändert worden ist.

*EM\_GETMODIFY*

wParam = 0;

lParam = 0;

- A/R: Falls der Inhalt verändert worden ist, wird der Wert TRUE, falls der Wert nicht verändert worden ist, der Wert FALSE zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## *EM\_GETPASSWORDCHAR*

Die Nachricht *EM\_GETPASSWORDCHAR* wird gesendet, um das Zeichen zu bekommen, dass bei einer Passworteingabe benutzt wird.

EM\_GETPASSWORDCHAR

wParam = 0;

lParam = 0;

■ A/R: Als `return` wird das Zeichen zurückgeliefert.

N: Falls kein Zeichen existiert, wird der Wert `NULL` zurückgeliefert.

Unterstützung:

■ WinNT 3.1 und später

■ Win95 und später

■ WinCE 1.0 und später

Header-Datei: `WINUSER.H`

*EM\_GETRECT*

Die Nachricht `EM_GETRECT` wird gesendet, um Informationen über die Größe des eingestellten Rechtecks zu bekommen.

EM\_GETRECT

wParam = 0;

lParam (LPARAM) (LPRECT) lprc;

■ E: `lParam` ist der Zeiger auf eine `RECT` Struktur.

Unterstützung:

■ WinNT 3.1 und später

■ Win95 und später

■ WinCE 1.0 und später

Header-Datei: `WINUSER.H`

*EM\_GETSEL*

Die Nachricht `EM_GETSEL` wird gesendet, um Anfang- und Endposition der Markierung herauszufinden.

EM\_GETSEL

wParam = (LPARAM) (LPDWORD) lpdwStart;

lParam = (LPARAM) (LPDWORD) lpdwEnd;

■ A: `wParam` ist ein Zeiger auf einen `WORD`-Wert, in dem die Startposition abgespeichert ist.

- A: `lParam` ist ein Zeiger auf einen WORD-Wert, in dem die Endposition abgespeichert ist.
- A/R: Die Positionen werden auch als `return` zurückgeliefert. Im unteren WORD ist die Startposition, im oberen WORD die Endposition. In Sonderfällen wird der Wert `-1` zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *EM\_GETTHUMB*

Die Nachricht `EM_GETTHUMB` wird gesendet, damit die Position der ScrollBox zurückgeliefert wird.

```
EM_GETTHUMB
wParam = 0;
lParam = 0;
```

- A/R: Als `return` wird die Position der ScrollBox zurückgeliefert.

Unterstützung:

- WinNT 3.51 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *EM\_GETWORDBREAKPROC*

Die Nachricht `EM_GETWORDBREAKPROC` wird gesendet, um den Zeiger auf die Funktion zu bekommen.

```
EM_GETWORDBREAKPROC
wParam = 0;
lParam = 0;
```

- A/R: Zurückgeliefert wird ein Zeiger auf eine Funktion.  
N: Falls die Funktion nicht existiert, wird der Wert `NULL` zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- nicht unterstützt

Header-Datei: WINUSER.H

*EM\_LIMITTEXT*

Die Nachricht *EM\_LIMITTEXT* wird gesendet, um die maximale Anzahl der Zeichen eines Edit-Feldes festzulegen.

*EM\_LIMITTEXT*

wParam = (WPARAM) cchMax;

lParam = 0;

- E: wParam gibt die maximale Anzahl der Zeichen an. Falls der Wert NULL ist, wird bei einzeiligen Edit-Felder der Wert auf 0x7FFFFFFE festgelegt. Bei mehrzeiligen Edit-Feldern wird der Wert auf 0xFFFFFFFF gesetzt.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- nicht unterstützt

Header-Datei: WINUSER.H

*EM\_LINEFROMCHAR*

Die Nachricht *EM\_LINEFROMCHAR* wird gesendet, um die Zeile in einem mehrzeiligen Edit-Feld herauszufinden, die einen bestimmten Zeichenindex enthält.

*EM\_LINEFROMCHAR*

wParam = (WPARAM) index;

lParam = 0;

- E: wParam gibt den Zeichenindex an. Falls dieser Wert -1 ist, wird die Position des Carets zurückgeliefert. Falls ein Bereich markiert ist, wird anstelle des Carets die Startposition des markierten Bereiches zurückgeliefert.

- A/R: Der Indexwert der Linie wird zurückgeliefert. Dieser Wert basiert auf NULL.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- nicht unterstützt

Header-Datei: WINUSER.H

### *EM\_LINEINDEX*

Die Nachricht *EM\_LINEINDEX* wird gesendet, um den Zeichenindex bis zum angegebenen Zeilenindex zu ermitteln.

*EM\_LINEINDEX*

wParam = (WPARAM) line;

lParam = 0;

- E: wParam ist der Zeilenindex, durch den der Zeichenindex zurückgeliefert wird. Ein Wert von -1 bestimmt die aktuelle Zeile.
- A/R: Zurückgeliefert wird der Wert des Zeichenindex. Dieser Wert ist -1, falls der angegebene Zeilenindex größer ist als der tatsächliche.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_LINELENGTH*

Die Nachricht *EM\_LINELENGTH* wird angegeben, um die Länge einer Zeile herauszufinden. Dazu wird ein Zeichenindex angegeben.

*EM\_LINELENGTH*

wParam = (WPARAM) index;

lParam = 0;

- E: wParam gibt den Zeichenindex an.
- A/R: Als return wird die Länge der Zeile zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

*EM\_LINESCROLL*

Die Nachricht *EM\_LINESCROLL* wird gesendet, um den Text horizontal oder vertikal zu scrollen.

*EM\_LINESCROLL*

wParam = (WPARAM) cxScroll;

lParam = (LPARAM) cyScroll;

- E: wParam gibt die Anzahl der Zeichen an, die horizontal gescrollt werden sollen.
- E: lParam gibt die Anzahl der Zeichen an, die vertikal gescrollt werden sollen.
- A/R: Falls die Nachricht an ein mehrzeiliges Edit-Feld gesendet wurde, wird der Wert TRUE zurückgeliefert. Falls der Wert an ein einzeliges Edit-Feld gesendet wurde, wird der Wert FALSE zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

*EM\_POSFROMCHAR*

Die Nachricht *EM\_POSFROMCHAR* wird gesendet, um durch die Angabe eines Zeichenindex eine Beschreibung der Koordinaten zu erhalten.

*EM\_POSFROMCHAR*

wParam = (WPARAM) Index;

lParam = 0;

- `wParam` ist der Index des Zeichens.
- `A/R`: Die Koordinaten des Zeichens werden zurückgeliefert.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *EM\_REPLACESEL*

Die Nachricht `EM_REPLACESEL` wird gesendet, um den markierten durch den angegebenen Text zu ersetzen.

`EM_REPLACESEL`

```
fCanUndo = (BOOL) wParam ;
lpszReplace = (LPCTSTR) lParam ;
```

- `E`: `wParam` gibt an, ob die Funktion wieder rückgängig gemacht werden kann. `TRUE` bedeutet, dass die Funktion wieder rückgängig, `FALSE`, dass die Funktion nicht wieder rückgängig gemacht werden kann.
- `E`: `lParam` gibt den Text an, der eingefügt werden soll.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *EM\_SCROLL/WM\_VSCROLL*

Die Nachricht `EM_SCROLL` wird gesendet, um den Text in einem mehrzeiligen Edit-Feld vertikal zu scrollen.

`EM_SCROLL`

```
wParam = (WPARAM) (INT) nScroll;
lParam = 0;
```

- `nScroll` gibt durch eine Konstante an, wie gescrollt werden soll.
  - `SB_LINEDOWN` bestimmt, dass eine Zeile weiter nach unten gescrollt werden soll.
  - `SB_LINEUP` bestimmt, dass eine Zeile weiter nach oben gescrollt werden soll.
  - `SB_PAGEDOWN` bestimmt, dass eine Seite weiter nach unten gescrollt werden soll.
  - `SB_PAGEUP` bestimmt, dass eine Seite weiter nach oben gescrollt werden soll.
- A/R: Falls die Funktion erfolgreich ausgeführt wurde, wird der Wert `TRUE` im oberen `WORD` zurückgeliefert. Im unteren `WORD` wird die Anzahl der tatsächlich gescrollten Zeilen zurückgeliefert. Falls ein Fehler auftritt, wird der Wert `FALSE` zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

*EM\_SCROLLCARET*

Die Nachricht `EM_SCROLLCARET` wird benutzt, um den Caret in das Edit-Feld zu holen.

`EM_SCROLLCARET`

`wParam = 0 ;`

`lParam = 0 ;`

- A/R: Falls die Nachricht an ein Edit-Feld gesendet wurde, wird ein Wert ungleich `NULL` zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später



Header-Datei: WINUSER.H

### *EM\_SETHANDLE*

Die Nachricht *EM\_SETHANDLE* wird gesendet, um den Handle des Speichers zu setzen, der den Text eines mehrzeiligen Edit-Feldes beinhaltet.

*EM\_SETHANDLE*

wParam = (WPARAM) (HLOCAL) hLoc;

lParam = 0;

- E: wParam ist der Handle auf den Speicher, der den Text beinhalten soll.

Unterstützung:

- WinNT 3.1 und später
- Win95 und Win98 nicht unterstützt
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_SETLIMITTEXT*

Die Nachricht *EM\_SETLIMITTEXT* wird gesendet, um die maximale Textlänge anzugeben.

*EM\_SETLIMITTEXT*

wParam = (WPARAM) Max;

lParam = 0;

- E: wParam ist die maximale Länge des Texts.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_SETMARGINS*

Die Nachricht *EM\_SETMARGINS* wird gesendet, um die Breite der Ränder eines Edit-Felds anzugeben.

EM\_SETMARGINS

wParam = (WPARAM) Margin;

lParam = (LPARAM) MAKELONG(wLeft, wRight);

- E: wParam gibt an, welcher Rand neu anzugeben ist. Dazu werden Konstanten benutzt.

EC\_LEFTMARGIN bestimmt, dass die Größe des linken Randes neu gesetzt werden soll.

EC\_RIGHTMARGIN bewirkt, dass die Größe des rechten Randes neu gesetzt werden soll.

EC\_USEFONTINFO gibt an, dass die Größe der Ränder durch die Schriftart gesetzt wird. Einzeilige Edit-Felder setzen die Ränder dann auf die durchschnittliche Zeichenbreite. Mehrzeilige Edit-Felder setzen den rechten Rand durch den Buchstaben A und den linken Rand durch den Buchstaben C.

- lParam gibt den Wert des linken und rechten Randes an.

Unterstützung:

- WinNT 4.0 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

*EM\_SETMODIFY*

Die Nachricht EM\_SETMODIFY wird gesendet, um anzugeben, ob das Edit-Feld modifiziert worden ist.

EM\_SETMODIFY

wParam = (WPARAM) Modified;

lParam = 0;

- wParam gibt an, ob der Text modifiziert worden ist. Wenn dieser Wert TRUE ist, ist der Text verändert worden, ist er FALSE, entsprechend nicht modifiziert worden.

Unterstützung:

- WinNT 3.1 und später

- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_SETPASSWORDCHAR*

Die Nachricht *EM\_SETPASSWORDCHAR* wird gesendet, um das Passwortzeichen zu setzen.

*EM\_SETPASSWORDCHAR*

```
wParam = (WPARAM) (UINT) Zeichen;
lParam = 0;
```

- E: *wParam* gibt das Zeichen an.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_SETREADONLY*

Die Nachricht *EM\_SETREADONLY* wird gesendet, um *ES\_READONLY* eines Edit-Felds zu setzen oder nicht zu setzen.

*EM\_SETREADONLY*

```
wParam = (WPARAM) (BOOL) ReadOnly;
lParam = 0;
```

- E: *ReadOnly* bestimmt, dass *ES\_READONLY* gesetzt wird bzw. nicht gesetzt wird. Falls der Wert *TRUE* ist, wird *ES\_READONLY* gesetzt, falls *FALSE*, nicht gesetzt.
- A/R: Falls die Funktion erfolgreich ausgeführt wird, liefert die Funktion einen Wert ungleich *NULL* zurück.  
N: Falls ein Fehler auftritt, liefert die Funktion den Wert *NULL* zurück.

Unterstützung:

- WinNT 3.1 und später

- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_SETRECT*

Die Nachricht *EM\_SETRECT* wird gesendet, um das formatierte Rechteck eines Edit-Felds zu setzen. Das Edit-Feld wird danach neu gezeichnet.

*EM\_SETRECT*

wParam = 0;

lParam = (LPARAM) (LPRECT) lprc;

- E: *lParam* ist ein Zeiger auf eine *RECT* Struktur. Die Struktur gibt die Größe des Rechtecks an.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### *EM\_SETRECTNP*

Die Nachricht *EM\_SETRECTNP* wird gesendet, um das formatierte Rechteck eines Edit-Felds zu setzen. Das Edit-Feld wird danach nicht neu gezeichnet.

*EM\_SETRECTNP*

wParam = 0;

lParam = (LPARAM) (LPRECT) lprc;

- E: *lParam* ist ein Zeiger auf eine *RECT*-Struktur, die die Größe des Rechtecks angibt.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## EM\_SETSEL

Die Nachricht EM\_SETSEL wird gesendet, um die Anzahl der markierten Zeichen eines Edit-Felds zu setzen.

EM\_SETSEL

wParam = (WPARAM) (INT) nStart;

lParam = (LPARAM) (INT) nEnd;

- E: wParam bestimmt die Startposition.
- E: lParam bestimmt die Endposition.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## EM\_SETTABSTOPS

Die Nachricht EM\_SETTABSTOPS wird gesendet, um die Tabstops in einem mehrzeiligen Edit-Feld zu setzen.

EM\_SETTABSTOPS

wParam = (WPARAM) Tabs;

lParam = (LPARAM) (LPDWORD) lpdwTabs;

- E: wParam bestimmt die Anzahl der Tabstops.
- E: lParam bestimmt die Tabstops.
- A/R: Falls die Funktion erfolgreich ausgeführt wurde, liefert die Funktion den Wert TRUE zurück, falls bei der Ausführung der Funktion ein Fehler auftrat, den Wert FALSE.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

Weitere unterstützte Nachrichten

WM\_COPY

WM\_CUT

WM\_PASTE

### Nachrichten vom Edit-Feld an das Parent-Fenster

WM\_COMMAND

Die WM\_COMMAND Nachricht wird dazu benutzt, das übergeordnete Fenster über Ereignisse zu informieren. Die Nachricht WM\_COMMAND wird nicht nur von Steuerelementen genutzt, sondern auch von Menüs oder von besonderen Tasten.

WM\_COMMAND

```
wNotifyCode = HIWORD(wParam);
```

```
wID = LOWORD(wParam);
```

```
hwndCtl = (HWND) lParam;
```

- A: `wNotifyCode` bestimmt die Nachricht genauer. Hier wird die Notification-Nachricht angegeben.
- A: `wID` bestimmt die ID des Steuerelements. Diese ID wurde in der Funktion `CreateWindow` durch `hMenu` angegeben.
- A: `hwndCtl` bestimmt, von welchem Steuerelement die Nachricht kam.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

WM\_CTLCOLOREDIT

Die Nachricht WM\_CTLCOLOREDIT wird gesendet, damit das Parent-Fenster die Farbe des Edit-Felds ändern kann.

WM\_CTLCOLOREDIT

```
hdcEdit = (HDC) wParam;
```

```
hwndEdit = (HWND) lParam;
```

- A: `hdcEdit` bestimmt den Gerätekontext des Edit-Feldes.
- A: `hwndEdit` bestimmt das Fensterobjekt des Edit-Feldes.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 2.0 und später

Header-Datei: `WINUSER.H`

### Notification-Nachrichten

`EN_CHANGE` bestimmt, dass der Text des Edit-Felds geändert wurde. Das untere `WORD` von `wParam` gibt die ID des Edit-Felds an. `lParam` bestimmt den Handle des Fensters.

`EN_SETFOCUS` bestimmt, dass das Edit-Feld den Eingabefokus erhalten hat. Das untere `WORD` von `wParam` ist die ID des Edit-Feldes. `lParam` ist der Handle auf das Edit-Feld.

`EN_UPDATE` bestimmt, dass das Edit-Feld neu gezeichnet werden soll. Diese Nachricht wird gesendet, wenn der Text im Edit-Feld geändert wurde.

`EN_HSCROLL` gibt an, dass der Benutzer auf den horizontalen Scrollbalken geklickt hat. Diese Nachricht wird gesendet, bevor das Edit-Feld neu gezeichnet wird.

`EN_VSCROLL` erklärt, dass der Benutzer auf den vertikalen Scrollbalken geklickt hat. Diese Nachricht wird gesendet, bevor das Edit-Feld neu gezeichnet wird.

`EN_MAXTEXT` sagt aus, dass der eingefügte Text zu groß ist.

`EN_KILLFOCUS` bestimmt, dass das Edit-Feld nicht mehr den Eingabefokus hat.

`EN_ERRSPACE` bestimmt, dass nicht genügend Speicher zur Verfügung steht, um eine Aktion auszuführen.

### 19.1.3 ListBox

Eine ListBox ist ein Steuerelement, das eine Liste von Elementen enthält. Aus dieser Liste von Elementen kann der Benutzer eines auswählen. In diesem Kapitel werden nur die Grundlagen der ListBox besprochen.

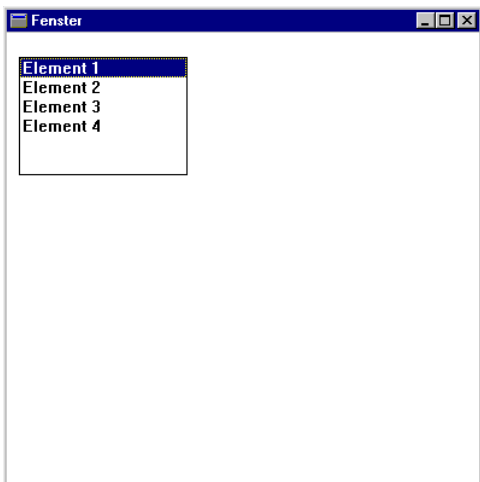


Bild 19.3: Eine ListBox

#### Styles

LBS\_STANDARD bestimmt, dass die Elementen in der ListBox alphabetisch sortiert werden. Das Parent-Fenster empfängt eine Eingabe, falls ein Klick oder ein Doppelklick ein Element bestimmen.

#### Nachrichten vom Parent-Fenster an die ListBox

##### LB\_GETCURSEL

Die Nachricht LB\_GETCURSEL wird gesendet, um den Index des gerade markierten Elementes zu bekommen.

```
LB_GETCURSEL  
wParam = 0;  
lParam = 0;
```



- A/R: Der Index des markierten Elementes wird zurückgeliefert. Dieser Index basiert auf dem Wert `o`. Falls kein Eintrag markiert ist, wird der Wert von `LB_ERR` zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *LB\_GETCOUNT*

Die Nachricht `LB_GETCOUNT` wird gesendet, um die Anzahl der Elemente in einer `Listbox` zu bekommen.

`LB_GETCOUNT`

`wParam = 0;`

`lParam = 0;`

- A/R: Falls die Funktion erfolgreich ausgeführt wird, liefert sie den Wert der Elemente in der `Listbox` zurück. Falls ein Fehler auftritt, wird der Wert `LB_ERR` zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: `WINUSER.H`

### *LB\_ADDSTRING*

Die Nachricht `LB_ADDSTRING` wird gesendet, um einer `Listbox` einen String zuzuweisen. Wenn `LBS_SORT` nicht gesetzt ist, wird der String am Ende der Liste eingefügt.

`LB_ADDSTRING`

`wParam = 0;`

`lParam = (LPARAM) (LPCTSTR) lpsz;`

- E: `lParam` ist der Text des neuen Elementes.

- A/R: Der Wert des Index des String wird zurückgeliefert. Falls ein Fehler auftritt, wird der Wert LB\_ERR zurückgeliefert. Falls nicht genug Platz ist, um den String abzuspeichern wird der Wert LB\_ERRSPACE zurückgeliefert.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

*LB\_DELETESTRING*

Die Nachricht LB\_DELETESTRING wird gesendet, um einen String aus einer ListBox zu entfernen.

LB\_DELETESTRING

wParam = (WPARAM) index;

lParam = 0;

- E: wParam bestimmt den Index des Strings, der entfernt werden soll.
- A/R: Die Anzahl der Strings, die noch in der Liste verbleiben, wird zurückgeliefert. Falls ein Fehler passiert, wird der Wert LB\_ERR zurückgegeben.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

## Nachrichten von der ListBox an das Parent-Fenster

### WM\_COMMAND

Die Nachricht WM\_COMMAND wird dazu benutzt, das übergeordnete Fenster über Ereignisse zu informieren. Sie wird nicht nur von Steuerelementen, sondern auch von Menüs oder von besonderen Tasten genutzt.

```
WM_COMMAND  
wNotifyCode = HIWORD(wParam);  
wID = LOWORD(wParam);  
hwndCtl = (HWND) lParam;
```

- A: `wNotifyCode` bestimmt die Nachricht genauer. Hier wird die Notification-Nachricht angegeben.
- A: `wID` bestimmt die ID des Steuerelements. Diese ID wurde in der Funktion `CreateWindow` durch `hMenu` angegeben.
- A: `hwndCtl` bestimmt, von welchem Steuerelement die Nachricht kam.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: WINUSER.H

### WM\_CTLCOLORLISTBOX

Die Nachricht WM\_CTLCOLORLISTBOX wird gesendet, damit das Parent-Fenster die Farbe der ListBox ändern kann.

```
WM_CTLCOLORLISTBOX  
hdcListBox = (HDC) wParam;  
hwndListBox = (HWND) lParam;
```

- A: `hdcListBox` bestimmt den Gerätekontext der ListBox.
- A: `hwndListBox` bestimmt das Fensterobjekt der ListBox.

Unterstützung:

- WinNT 3.1 und später

- Win95 und später
- WinCE 2.0 und später

Header-Datei: WINUSER.H

## Notification-Nachrichten

LBN\_DBLCLICK bestimmt, dass ein Doppelklick auf ein Element in der ListBox stattgefunden hat. Diese Nachricht wird nur durch LBS\_NOTIFY gesendet.

### 19.1.4 Static

#### Allgemeines

Ein Steuerelement *Static* ist ein Textfeld. Ein Textfeld gibt Text aus. Dieser Text bleibt im Gegensatz zu der einfachen Funktion `TextOut` beim Verschieben des Fensters erhalten.



Bild 19.4: Ein Textfeld

#### Styles

SS\_SIMPLE bestimmt, dass ein einfaches Rechteck erstellt wird, in dem ein Text angezeigt wird, der links ausgerichtet ist.

## Nachrichten vom Parent-Fenster an das Textfeld

Es gibt keine wichtigen Nachrichten.

## Nachrichten vom Textfeld an das Parent-Fenster

### *WM\_COMMAND*

Die Nachricht *WM\_COMMAND* wird dazu genutzt, das übergeordnete Fenster über Ereignisse zu informieren. Sie wird nicht nur von Steuerelementen, sondern auch von Menüs oder von besonderen Tasten verwendet.

```
WM_COMMAND
wNotifyCode = HIWORD(wParam);
wID = LOWORD(wParam);
hwndCtl = (HWND) lParam;
```

- **A:** *wNotifyCode* bestimmt die Nachricht genauer. Hier wird die Notification-Nachricht angegeben.
- **A:** *wID* bestimmt die ID des Steuerelements. Diese ID wurde in der Funktion *CreateWindow* durch *hMenu* angegeben.
- **A:** *hwndCtl* bestimmt, von welchem Steuerelement die Nachricht kam.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 1.0 und später

Header-Datei: *WINUSER.H*

### *WM\_CTLCOLORSTATIC*

Die Nachricht *WM\_CTLCOLORSTATIC* wird gesendet, damit das Parent-Fenster die Farbe des Textfeldes ändern kann.

```
WM_CTLCOLORSTATIC
hdcStatic = (HDC) wParam;
hwndStatic = (HWND) lParam;
```

- **A:** *hdcStatic* bestimmt den Gerätekontext des Textfeldes.
- **A:** *hwndStatic* bestimmt das Fensterobjekt des Textfeldes.

Unterstützung:

- WinNT 3.1 und später
- Win95 und später
- WinCE 2.0 und später

Header-Datei: WINUSER.H

### **Notification-Nachrichten**

STN\_CLICKED bestimmt, dass einmal auf das Textfeld geklickt wurde.

STN\_DBLCLK bestimmt, dass mit einem Doppelklick auf das Textfeld geklickt wurde.

TEIL III

Nitty  
Nitty  
Gritty

GO AHEAD!





## **20.1 Allgemeines**

DirectX ist eine neue API, die Grafik- und Soundfunktionen bereitstellt. Diese Grafikfunktionen sind sehr schnell, da sie einen besseren Zugriff auf die Hardware ermöglichen. DirectX ermöglicht ein geräte-unabhängiges Programmieren und ein besseres Ansteuern der Hardware. Und durch den besseren Hardwarezugriff wird das Entwickeln von Spielen mit Grafik und Sound, die sehr hohe Ansprüche an die Leistung eines Systems stellen, erleichtert.

Die API von DirectX ist mit C++-Objekten aufgebaut (s. auch COM). Wir müssen also zwischen Windows- und C++-Objekten unterscheiden.

Wir schauen uns in diesem Kapitel ein komplexeres Beispielprogramm an, das die Grundeigenschaften von DirectX beschreibt.

## **20.2 Ein DirectX-Programm**

### **20.2.1 Allgemeines**

Dieses Programm will verdeutlichen, wie schnelle Grafiken ohne Fehler erzeugt werden. Dieses Verfahren wird bei fast allen modernen Spielen angewandt. Es wird ein Speicher erstellt, der die gesamte Grafik aufnimmt, in diesen Speicher wird ausschließlich gezeichnet. Dann wird der ganze Speicher in den Bildschirmspeicher überschrieben. Falls alle Grafiken identisch übertragen werden, kommt es zu Verzögerungen. Die letzte Grafik hat immer die geringste Verweildauer im Bildschirmspeicher.

---

Primary Buffer



Bild 20.1: Die Anwendung nach dem Start

### 20.2.2 Quelltext

```
#include <windows.h>
#include <ddraw.h>

LPDIRECTDRAW7 lpDD=NULL;
LPDIRECTDRAWSURFACE7 lpDDSPimary=NULL;
LPDIRECTDRAWSURFACE7 lpDDSSBack=NULL;

int Reihe = 1;

LRESULT CALLBACK WindowProc(HWND, unsigned,
                             WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
```

```

        int         nCmdShow)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = sizeof(DWORD);
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)
        GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "WinProg";

    RegisterClass(&wc);

    int ScreenBreite = GetSystemMetrics(SM_CXSCREEN);
    int ScreenHoehe = GetSystemMetrics(SM_CYSCREEN);

    HWND hWnd;

    hWnd = CreateWindow("WinProg",
        "DirectX Programm",
        WS_POPUP,
        0,
        0,
        ScreenBreite,
        ScreenHoehe,
        NULL,
        NULL,
        hInstance,
        NULL);

    ShowWindow(hWnd, nCmdShow);

    UpdateWindow(hWnd);

```

```

ShowCursor (false);

SetTimer (hWnd, NULL, 1000, NULL);

DirectDrawCreateEx(NULL, (void **) &lpDD,
                   IID_IDirectDraw7, NULL);
lpDD->SetCooperativeLevel(hWnd, DDSCL_EXCLUSIVE |
                          DDSCL_FULLSCREEN );
lpDD->SetDisplayMode( 640, 480, 16, 0, 0);

DDSURFACEDESC2 ddsd;
ZeroMemory(&ddsd,sizeof(ddsd));
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                     DDSCAPS_FLIP |
                     DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;

lpDD->CreateSurface( &ddsd, &lpDDSPPrimary, NULL );

DDSCAPS2 ddscaps;
ZeroMemory(&ddscaps,sizeof(ddscaps));
ddscaps.dwCaps=DDSCAPS_BACKBUFFER;

lpDDSPPrimary->GetAttachedSurface(&ddscaps,&lpDDSBBack);

MSG msg;

while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}

lpDDSBBack->Release ();
lpDDSPPrimary->Release ();

```

```

lpDD->Release ();

ShowCursor (true);

return msg.wParam;

}

LRESULT CALLBACK WindowProc(HWND hWnd,
                             unsigned uMsg,
                             WPARAM wParam,
                             LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_TIMER:
            if (Reihe == 1)
            {
                HDC hdc;
                lpDDSBack->GetDC (&hdc);
                TextOut (hdc, 10, 10,
                        "Primary Buffer",
                        lstrlen("Primary Buffer"));
                lpDDSBack->ReleaseDC (hdc);
                Reihe = 2;
            }
            else
            {
                HDC hdc;
                lpDDSBack->GetDC (&hdc);
                TextOut (hdc, 10, 10, "Back Buffer",
                        lstrlen("Back Buffer"));
                lpDDSBack->ReleaseDC (hdc);
                Reihe = 1;
            }
            lpDDSPPrimary->Flip(0,DDFLIP_WAIT);
            return 0;
        case WM_KEYDOWN:

```

```

        DestroyWindow (hWnd);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, uMsg,
                               wParam, lParam);
    }
return 0;
}

```

### 20.2.3 Beschreibung

#### Initialisierung

```

DirectDrawCreateEx(NULL, (void **) &lpDD,
                   IID_IDirectDraw7, NULL);
lpDD->SetCooperativeLevel(hWnd, DDSCL_EXCLUSIVE |
                          DDSCL_FULLSCREEN );
lpDD->SetDisplayMode( 640, 480, 16, 0, 0);

DDSURFACEDESC2 ddsd;
ZeroMemory(&ddsd,sizeof(ddsd));
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                     DDSCAPS_FLIP |
                     DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;

lpDD->CreateSurface( &ddsd, &lpDDSPimary, NULL );

DDSCAPS2 ddscaps;
ZeroMemory(&ddscaps,sizeof(ddscaps));
ddscaps.dwCaps=DDSCAPS_BACKBUFFER;

lpDDSPimary->GetAttachedSurface(&ddscaps,&lpDDSBck);

```

Zuerst wird ein DirectDraw-Objekt angelegt.

Dieses Objekt wird dann mit dem Fenster verbunden und der Vollbildmodus wird aktiviert.

Die Einstellungen der Grafikkarte werden vorgenommen (Auflösung, Farbe etc.).

Das erste Surface-Objekt wird angelegt, es ist der Bildschirmspeicher. Dieses wird dann auch gleich mit dem DirectDraw-Objekt verbunden.

Nun wird zweite Surface-Objekt erstellt. Dies ist ein einfaches Speicherobjekt, in das gezeichnet wird. Es wird mit dem ersten Surface-Objekt verbunden.

### Der Hauptteil

```
if (Reihe == 1)
{
    HDC hdc;
    lpDDSBack->GetDC (&hdc);
    TextOut (hdc, 10, 10,
            "Primary Buffer",
            lstrlen("Primary Buffer"));
    lpDDSBack->ReleaseDC (hdc);
    Reihe = 2;
}
else
{
    HDC hdc;
    lpDDSBack->GetDC (&hdc);
    TextOut (hdc, 10, 10, "Back Buffer",
            lstrlen("Back Buffer"));
    lpDDSBack->ReleaseDC (hdc);
    Reihe = 1;
}
lpDDSPPrimary->Flip(0,DDFLIP_WAIT);
```

Im zweiten Surface-Speicher wird gezeichnet. Es wird entweder „Primary Buffer“ oder „Back Buffer“ hineingeschrieben. Die alten GDI-Funktionen können also noch verwendet werden. Ein Gerätekontext kann auch von DirectX-Objekten zurückgeliefert werden.

Der gesamte erste Surfacespeicher wird vom zweiten Surfacespeicher überschrieben. Es wurden Bitmaps aus den vorherigen Kapiteln benutzt.

## Deinitialisierung

```
1pDDSSBack->Release ();  
    1pDDSPPrimary->Release ();  
    1pDD->Release ();
```

Alle Objekte werden nun aus dem Speicher freigegeben. Zuerst muss das abhängigste Objekt freigegeben werden. Dies muss unbedingt in dieser Reihenfolge passieren.

## 20.3 DirectX und Bitmaps

### 20.3.1 Allgemeines

Dieses Programm demonstriert, wie man Bitmaps mit DirectX benutzt.

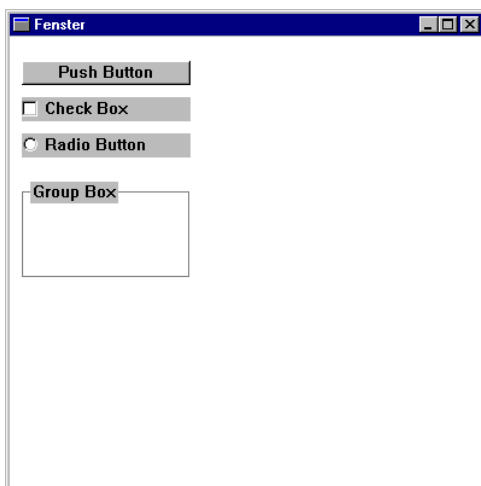


Bild 20.2: Die Anwendung nach dem Start



### 20.3.2 Quelltext

```
#include <windows.h>
#include <ddraw.h>

LPDIRECTDRAW7 lpDD=NULL;
LPDIRECTDRAWSURFACE7 lpDDSPPrimary=NULL;
LPDIRECTDRAWSURFACE7 lpDDSPBack=NULL;
LPDIRECTDRAWSURFACE7 lpBitmap=NULL;
LPDIRECTDRAWSURFACE7 lpBitmapp=NULL;

int Reihe = 1;

LRESULT CALLBACK WindowProc(HWND, unsigned,
                             WPARAM, LPARAM);
LPDIRECTDRAWSURFACE7 bitmap_surface(LPCTSTR);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
    WNDCLASS wc;
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = sizeof(DWORD);
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)
        GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "WinProg";
    RegisterClass(&wc);

    int ScreenWidth = GetSystemMetrics(SM_CXSCREEN);
```

```

int ScreenHeight = GetSystemMetrics(SM_CYSCREEN);

HWND hWnd;

hWnd = CreateWindow("WinProg",
                   "DirectX Programm",
                   WS_VISIBLE | WS_POPUP,
                   0,
                   0,
                   ScreenWidth,
                   ScreenHeight,
                   NULL,
                   NULL,
                   hInstance,
                   NULL);

ShowWindow(hWnd, nCmdShow);

UpdateWindow(hWnd);

ShowCursor (false);

SetTimer (hWnd, NULL, 1000, NULL);

DirectDrawCreateEx(NULL, (void **) &lpDD,
                   IID_IDirectDraw7, NULL);

lpDD->SetCooperativeLevel(hWnd, DDSCL_EXCLUSIVE |
                          DDSCL_FULLSCREEN );
lpDD->SetDisplayMode( 640, 480, 16, 0, 0);

DDSURFACEDESC2 ddsd;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                      DDSCAPS_FLIP |
                      DDSCAPS_COMPLEX;

```

```

ddsd.dwBackBufferCount = 1;

lpDD->CreateSurface( &ddsd, &lpDDSPPrimary, NULL );

DDSCAPS2 ddscaps;
ZeroMemory(&ddscaps, sizeof(ddscaps));
ddscaps.dwCaps=DDSCAPS_BACKBUFFER;

lpDDSPPrimary->GetAttachedSurface(&ddscaps,&lpDDSPBack);

lpBitmap = bitmap_surface ("bitmap.bmp");
lpBitmapp = bitmap_surface ("bitmapp.bmp");

MSG msg;

while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}

lpBitmapp->Release ();
lpBitmap->Release ();
lpDDSPBack->Release ();
lpDDSPPrimary->Release ();
lpDD->Release ();

ShowCursor (true);

return msg.wParam;

```

}

```
LRESULT CALLBACK WindowProc(HWND hWnd, unsigned uMsg,  
                             WPARAM wParam, LPARAM lParam)
```

```
{  
    switch (uMsg)  
    {  
        case WM_TIMER:  
            RECT rects;  
            rects.top = 0;  
            rects.left = 0;  
            rects.right = 639;  
            rects.bottom = 479;  
            if (Reihe == 1)  
            {  
                lpDDSSBack->BlitFast (0,0,lpBitmap,&rects,  
                                     DDBLTFAST_NOCOLORKEY);  
  
                Reihe = 2;  
            }  
            else  
            {  
                lpDDSSBack->BlitFast (0,0,lpBitmapp,&rects,  
                                     DDBLTFAST_NOCOLORKEY);  
  
                Reihe = 1;  
            }  
            lpDDSPPrimary->Flip(0,DDFLIP_WAIT);  
            return 0;  
        case WM_KEYDOWN:  
            DestroyWindow (hWnd);  
            return 0;  
        case WM_DESTROY:  
            PostQuitMessage(0);  
            break;  
        default:  
            return DefWindowProc(hWnd, uMsg,
```

```

        wParam, lParam);
    }
    return 0;
}

LPDIRECTDRAW_SURFACE7 bitmap_surface(LPCTSTR file_name)
{
    HDC hdc;
    HBITMAP bit;
    LPDIRECTDRAW_SURFACE7 surf;

    bit=(HBITMAP)
    LoadImage(NULL,
               file_name,
               IMAGE_BITMAP,
               0,0,
               LR_DEFAULTSIZE | LR_LOADFROMFILE);

    if (!bit) return NULL;
    BITMAP bitmap;
    GetObject( bit, sizeof(BITMAP), &bitmap );
    int surf_width=bitmap.bmWidth;
    int surf_height=bitmap.bmHeight;

    HRESULT ddrval;
    DDSURFACEDESC2 ddsd;
    ZeroMemory(&ddsd,sizeof(ddsd));
    ddsd.dwSize = sizeof(DDSURFACEDESC2);
    ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
    ddsd.ddsCaps.dwCaps =
    DDSCAPS_OFFSCREENPLAIN|DDSCAPS_SYSTEMMEMORY;
    ddsd.dwWidth = surf_width;
    ddsd.dwHeight = surf_height;
}

```

```
ddrval=lpDD->CreateSurface(&ddsd,&surf,NULL);
```

```
if (ddrval!=DD_OK)  
{
```

```
    DeleteObject(bit);  
    return NULL;
```

```
}  
else
```

```
{  
    surf->GetDC(&hdc);
```

```
    HDC bit_dc=CreateCompatibleDC(hdc);
```

```
    SelectObject(bit_dc,bit);
```

```
    BitBlt(hdc,0,0,  
           surf_width,surf_height,  
           bit_dc,0,0,SRCCOPY);
```

```
    surf->ReleaseDC(hdc);  
    DeleteDC(bit_dc);
```

```
}
```

```
    DeleteObject(bit);
```

```
return surf;  
}
```

20

# 21 UNICODE

---

## 21.1 Allgemeines

UNICODE ist der Nachfolger von ANSI. UNICODE ist ein 16-Bit-Zeichensatz, der 65536 verschiedene Zeichen in einem Zeichensatz ermöglicht. In den ca. ersten 30000 Zeichen sind alle Zeichen der Welt vorhanden. Die Win32-API enthält Funktionen, die jeweils UNICODE und ANSI-Code unterstützen. So ist z. B. die Funktion `CreateWindow` als Funktion `CreateWindowA` und `CreateWindowW` vorhanden. Die Auswahl einer Funktion geschieht im Hintergrund durch eine Voreinstellung.

## 21.2 Betriebssysteme

### 21.2.1 Windows 95

Das System arbeitet mit ANSI. Der Einsatz von ANSI wird empfohlen.

### 21.2.2 Windows 98

Das System arbeitet mit ANSI. Es ist möglich, UNICODE zu benutzen. Der Einsatz von ANSI wird empfohlen.

### 21.2.3 Windows NT

Man kann ANSI verwenden, doch dies verlangsamt das System, da es mit UNICODE arbeitet. Jeder Text muss vorher umgewandelt werden. Der Einsatz von UNICODE wird empfohlen.

---

#### 21.2.4 Windows 2000

Man kann ANSI verwenden, doch dies verlangsamt das System, da es mit UNICODE arbeitet. Jeder Text muss vorher umgewandelt werden. Der Einsatz von UNICODE wird empfohlen.

#### 21.2.5 Windows CE

Man muss mit UNICODE arbeiten, denn das System arbeitet nur mit UNICODE.



## 22.1 Allgemeines

Dieses Kapitel soll keine Anleitung zum Programmieren sein, sondern es beschreibt, was COM ist, und wozu es genutzt wird.

COM ist ein Objektstandard und bedeutet *Component Object Model* (Modell für die Entwicklung von Objektkomponenten. COM wird auch als binärer Standard bezeichnet. Es ist ein Standard für Objekte. Dieser Standard kann von fast allen Sprachen genutzt werden (C++, Pascal, Basic usw.). Daten eines COM-Objekts werden durch die Methoden des COM-Objekts verändert. Die Sammlung von Methoden heißt Interface. Es gibt eine Standard-API. Diese API enthält Funktionen, mit denen man ein COM-Objekt erstellt und einen Zeiger auf COM-Objekte bekommt, der wie ein richtiges C++-Objekt genutzt wird. Ein COM-Objekt wird durch eine ID erstellt, die angibt, was für ein COM-Objekt erstellt werden soll.

## 22.2 DirectX

Auch DirectX benutzt COM-Objekte. Deswegen sucht man immer zuerst nach der ID, um einen Zeiger auf ein Objekt zu bekommen. Alle DirectX-Versionen werden überschrieben und auch die alten Programme laufen mit den neuen DirectX-Dateien, zu denen einfach neue Objekte hinzugefügt werden. Ein Beispiel hierfür ist DirectDraw. Es gibt verschiedene DirectDraw-Objekte:

1. IDirectDraw
2. IDirectDraw2
3. IDirectDraw4
4. IDirectDraw7

Durch COM kann DirectX von C++, Pascal und Visual Basic eingesetzt werden.

Vor COM war es nur möglich, Funktionen zu schreiben, die von anderen Sprachen genutzt werden konnten. Durch COM ist es jetzt auch möglich, Objekte zu schreiben, die von anderen Sprachen auch genutzt werden.

## 23.1 Allgemeines

Ressourcen sind Daten, die in der EXE-Datei oder der DLL-Datei gespeichert sind. Diese Daten sind nicht Bestandteil der Anwendung, sondern zusätzliche Daten, die der Anwendung mitgegeben werden.

Diese Daten werden durch eine Datei im Textformat beschrieben („*resource definition file*“). Diese Datei enthält Verweise auf die einzelnen Icons, Bitmaps, usw. Durch einen Ressourcencompiler werden diese Daten in einer einzigen binären Datei zusammengefasst. Diese wird der EXE- oder der DLL-Datei angefügt.

Ressourcen werden auf zwei Arten genutzt. Die Anwendung kann sie einerseits selber verwenden und aus ihnen Objekte machen. Alle Ressourcen haben einen Wert, der sie eindeutig identifiziert. Diesen Wert kann man aus der Datei RESOURCE.H ermitteln. Ressourcen werden aber andererseits auch von anderen Anwendungen benutzt. Ein gutes Beispiel dafür ist die Anzeige von Dateien. Dort wird eine Datei mit einem Symbol angezeigt. Dieses Symbol ist als Ressource in der Datei abgespeichert.

Einige DLL-Dateien – wie z.B. Schriftdateien - bestehen nur aus Ressourcen.

## 23.2 Ein Beispiel für eine Ressource

### 23.2.1 Allgemeines

In diesem Programm wird ein Fenster erstellt. Dieses Fenster wird mit einer Iconressource und einer Cursorressource verbunden.

### Quelltext der Hauptdatei

```
#include <windows.h>
#include "resource.h"
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

```
int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    WNDCLASS WndClass;
    WndClass.style = 0;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.lpfWndProc = WndProc;
    WndClass.hInstance = hInstance;
    WndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    WndClass.hCursor = LoadCursor (hInstance,
                                   MAKEINTRESOURCE(IDC_CURSOR1));
    WndClass.hIcon = LoadIcon (hInstance,
                               MAKEINTRESOURCE(IDI_ICON1));
    WndClass.lpszMenuName = 0;
    WndClass.lpszClassName = "WinProg";

    RegisterClass(&WndClass);

    HWND hWindow;
    hWindow = CreateWindow("WinProg","Fenster",
                          WS_OVERLAPPEDWINDOW,
                          0,0,600,400,NULL,NULL,
                          hInstance, NULL);

    ShowWindow (hWindow, nCmdShow);

    UpdateWindow (hWindow);

    MSG Message;
    while (GetMessage(&Message, NULL, 0, 0))
    {
```

23

```

        TranslateMessage (&Message);
        DispatchMessage(&Message);
    }

    return (Message.wParam);
}

LRESULT CALLBACK WndProc (HWND hWnd, UINT uiMessage,
                          WPARAM wParam, LPARAM lParam)
{
    switch(uiMessage)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default:
            return DefWindowProc (hWnd, uiMessage,
                                   wParam, lParam);
    }
}

```

### Quelltext der Ressourcdatei

```

#define IDI_ICON1                101
#define IDC_CURSOR1              102

#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS

#define _APS_NEXT_RESOURCE_VALUE  129
#define _APS_NEXT_COMMAND_VALUE  32771
#define _APS_NEXT_CONTROL_VALUE   1000
#define _APS_NEXT_SYMED_VALUE     110
#endif
#endif

```

## Quelltext der Skriptdatei

```
#include "resource.h"

LANGUAGE LANG_GERMAN, SUBLANG_GERMAN

#pragma code_page(1252)

IDI_ICON1 ICON DISCARDABLE "icon1.ico"

IDC_CURSOR1 CURSOR DISCARDABLE "cursor1.cur"
```

### 23.2.2 Erläuterung

#### MAKEINTRESOURCE

Das Makro MAKEINTRESOURCE macht aus einem Integerwert einen Wert, der die Ressource bestimmt.

#### Laden von Ressourcen

Um eine Ressource mit den Funktionen `LoadIcon` und `LoadCursor` zu laden, muss der Instance-Handle angegeben werden.

23

# Stichwortverzeichnis

## A

ANSI 49, 60  
ANSI\_FIXED\_FONT 75  
ANSI\_VAR\_FONT 75  
Anwendungen 147  
ASCII-Code 49

## B

BeginPaint 41  
Bildschirm-Gerätekontext-Objekt 99  
BitBlt 106  
BITMAPFILEHEADER 112  
BITMAPINFO 112  
Bitmaps 97  
Bitmap-Schriften 49  
Blockgrafiken 49  
BMP-Datei 108  
BN\_CLICKED 59  
Borland C++ 5.5 15  
Brush-Objekt 39, 45, 47  
BS\_CHECKBOX 94  
BS\_GROUPBOX 94  
BS\_RADIOBUTTON 94  
BUTTON 59, 93

## C

CheckBoxes 94  
CloseHandle 145  
COLORREF 40  
COM 329  
Compiler 15  
Component Object Model 329  
CreateBitmap 105  
CreateCompatibleDC 105  
CreateFile 142  
CreateMenu 128

CreatePen 39  
CreateSolidBrush 39  
CreateThread 152  
CreateWindow 24, 59  
Cursor 37

## D

Datei 135  
    neu erstellen 141  
    öffnen 141  
Dateisysteme 135  
DDB 97  
DEAFULT\_GUI\_FONT 75  
DefWindowProc 31  
DeleteDC 107  
Device Dependent Bitmap 97  
Device Independent Bitmap 97  
DIB 97, 98, 108  
DirectX 313  
DispatchMessage 28  
DLL-Dateien 155  
DllMain 160  
DOCINFO 178  
DrawText 56  
Drucker 171  
Druckerstandarddialog 176  
Dynamic Link Library 155

## E

EDIT 73, 81  
Edit-Feld 73  
Ellipse 47  
EndDoc 179  
EndPage 178  
EndPaint 41, 48  
Erstellen des Fensters 24

EXE-Datei 147  
ExitWindowsEx 94

## **F**

Farbtiefe 98  
Fensterklasse 21  
FillRect 46

## **G**

GDI 33  
GDI-Schriften 49  
Geräteabhängiges Bitmap 97  
Gerätekontext-Objekt 33  
Geräteunabhängiges Bitmap 97  
GetClientRect 106  
GetCurrentDirectory 133  
GetDeviceCaps 132  
GetMessage 27  
GetStockObject 54  
GetTextMetrics 68  
GetWindowsDirectory 133  
Grafikausgabe 33  
Graphics Device Interface 33  
GroupBoxes 94

## **H**

HANDLE 22  
HFONT 54  
HMENU 128  
HWND 24

## **I**

Icon 37  
InsertMenuItem 128  
ISO-8859 49  
ISO-Standard 49

## **L**

Laden von Ressourcen 334  
LineTo 44  
LoadCursor 37

LoadIcon 37  
Istrcat 57  
Istrcmp 57, 84  
Istrcpy 56  
Istrlen 57

## **M**

Menü 117  
    erstellen 126  
MENUITEMINFO 129  
Message-Loop 26  
Message-Struktur 29  
MoveToEx 44  
MS Visual C++ 6.0 17  
MSG 29  
Multitasking 26  
Multi-Threading 147

## **N**

Nachricht WM\_COMMAND 59  
Nachrichtenabarbeitung 30  
nicht-proportionale Schrift 50  
NTFS 135

## **O**

Objekt 24, 31  
OEM\_FIXED\_FONT 76  
OEM-Zeichensatz 49  
Or 25

## **P**

PAINTSTRUCT 42  
Pen-Objekt 40, 45, 47  
PostQuitMessage 30, 31  
PRINTDLG 176  
PrintDlg 176  
Proportionale Schrift 50  
Protected-Fat-File-System 135  
Prototyp der Funktion 30  
Prozess 147



## R

RadioButtons 94  
ReadFile 143  
RECT 42  
Rectangle 45  
RegisterClass 23  
Resource definition file 331  
Ressourcen 331  
RGB-Farben 98  
RGB-Makro 40  
RoundRect 45  
Rückgabewert der Anwendung 30

## S

Scan-Codes 49, 60  
Schriftarten-Objekt 54  
SelectObject 43, 55  
SetBkColor 55, 56  
SetDIBitsToDevice 114  
SetMenu 131  
SetRect 46  
SetTextAlign 56  
SetTextColor 55  
SetTimer 166, 170  
ShowWindow 26  
Speicher-Gerätekontext-Objekt 99  
StartDoc 178  
StartPage 178  
Steuerelement 59  
    Edit-Feld 73  
    erstellen 64  
    WM\_COMMAND 66  
StretchBlt 107  
String-Funktionen 56  
switch 30  
Switch-Verzweigung 30  
SYSTEM\_FONT 60, 74  
System-Shutdown 85

## T

TerminateThread 153

Textausgabe 49  
TEXTMETRIC 69  
TextOut 55, 57  
Thread 147  
Timer 163  
Timer-Funktion 167  
Timer-Nachrichten 163  
TrueColor 98  
True-Type-Schriften 49

## U

UNICODE 327  
UpdateWindow 26

## V

Vektor-Schriftarten 50  
Verzeichnisse ermitteln 132  
Vordefinierte Fensterklassen 269  
Vordefinierte Schriftarten 54

## W

Win32-API 33  
    Dateiverwaltung 257  
    Datentypen 183  
    GDI 211  
    Windows-Grundlagen 187  
Windows.h 20  
WinMain 21  
WM\_COMMAND 66, 131  
WM\_CREATE 64  
WM\_DESTROY 30  
WM\_GETTEXT 82  
WM\_LBUTTONDOWN 104  
WM\_PAINT 39, 59  
WM\_QUIT 30  
WM\_RBUTTONDOWN 107  
WM\_TIMER 166  
WNDCLASS 21  
WndProc 30  
WriteFile 144  
WS\_CHILD 81