Win32-Programmierung: Text-Ausgabe

- Häufigste GDI-Funktion für Text-Ausgabe in Fenster:

```
hdc = GetDC (hwnd);
TextOut (hdc, x, y, psText, iLength);
ReleaseDC (hwnd, hdc);
```

hdc: Handle zum Gerätekontext (device context)

hwnd: Handle zum Empfänger-Fenster der Nachricht

x,y: Koordinaten der Startposition (Ursprung: oben links)

psText: Zeiger auf Zeichenkette der Ausgabe

iLength: Anzahl der auszugebenden Zeichen

  - **hdc**-Ermittlung über **`BeginPaint(hwnd,&ps)`** ergibt das z.Z. ungültige Rechteck, löscht es und kennzeichnet es als „gültig“ (bis **`WM_PAINT`**); muß mit **`EndPaint(hwnd,&ps)`** freigegeben werden.
  - **hdc**-Ermittlung über **`GetDC(hwnd)`** gibt d. gesamten Anwendungs-bereich zurück; mit **`ReleaseDC(hwnd,hdc)`** freizugeben; Löschg; keine „gültig“-Kennzeichnung (geeignet f. Abfragen, Zeicheneingabe)

Win32-Programmierung: Text-Ausgabe

Bemerkungen zu der vorausgegangenen Folie:

- „Gerätekontext“: Interne GDI-Datenstruktur mit 4-Byte-Handle (**`int`**), bezogen auf ein Ausgabegerät; dient der Koordination des Zugriffs auf dieses Gerät durch mehrere Prozesse und der Verbindung zum Anwendungsbereich eines Fensters (*client area*); enthält u.a. Codierung von Farbe für Text und Text-Hinterlegung, Schriftart und -größe, aktuelle Skalierung, aktuelles Clipping-Rechteck.

- Verwendet man **`GetDC()`** anstelle von **`BeginPaint()`** bei der Textausgabe (z.B. in **`HelloWin.c`**) – d.h.:

```
case WM_PAINT:
    hdc=GetDC(hwnd);   //statt: hdc=BeginPaint(hwnd, &ps);
    TextOut(hdc, x,y, psText, iLength);
    ReleaseDC(hwnd, hdc);     //statt EndPaint(hwnd, &ps);
```

  so bemerkt man ein leichtes Flimmern im Fenster: die Nicht-Kennzeichnung als „gültig“ bewirkt eine ständige Auffrischung durch Windows (ständiger Aufuf von **`WndProc()`** mit **`WM_PAINT`** – festzustellen z.B. mit Haltepunkt bei **`TextOut()`**).

Win32-Programmierung: Text-Ausgabe

- Erzwingung der Aktualisierung:

  `InvalidateRect (hwnd, NULL, TRUE);`

  Muß vor dem Aufruf von **`BeginPaint`** stehen. **`NULL`** steht für den gesamten Anwendungsbereich, **`TRUE`** für Löschen (**`FALSE`** für Unverändertlassen: Strukturvariable **`ps.fErase`**) des angegebenen (hier: gesamten) Bereichs .

- Unterbindung der Aktualisierung:

  `ValidateRect (hwnd, NULL);`

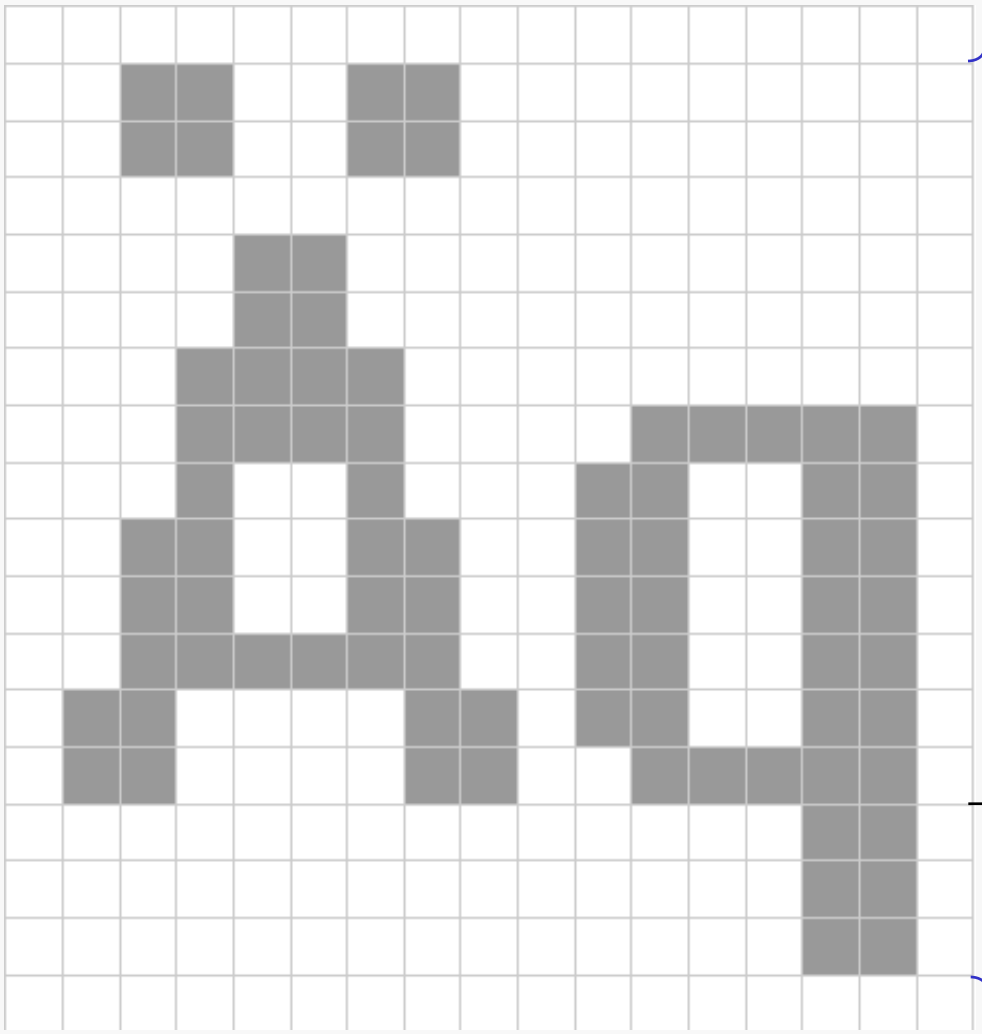
  (**`NULL`** wieder: im gesamten Bereich)

Win32-Programmierung: Text-Ausgabe

- Zur Text-Ausgabe werden Schrift-Maße benötigt (in Kombination mit gerätetechnischer Auflösung und Größe des Ausgabemediums).
- Notwendige (20) Angaben in Struktur **TEXTMETRIC** zusammengestellt (Defin.: **WINGDI.H**); darunter interessant:

```
typedef struct                    /*ungarische Notation:*/
{ LONG tmHeight;                /*Gesamthoehe –z.B.: 16 */
  LONG tmAscent;                          /*Oberlaenge*/
  LONG tmDescent;                        /*Unterlaenge*/
  LONG tmInternalLeading;       /*Platz fuer Akzente*/
  LONG tmExternalLeading;    /*Platz zw.Zeilen–z.B.0*/
  LONG tmAveCharWidth;  /*mittl.Breite Kleinbuchst.*/
  LONG tmMaxCharWidth;    /*max.Z.-Breite≈1,5*mittl.*/
                        /*... weitere 13 Angaben ...*/
} TEXTMETRIC;
```

Win32-Programmierung: Text-Ausgabe



*

Platz für Akzente u.ä.
(`tmInternalLeading`)

Oberlänge
(`tmAscent`)

Gesamthöhe
(`tmHeight`)

Grundlinie

Unterlänge
(`tmDescent`)

* *empfohlener Zeilenabstand
(`tmExternalLeading`)

Bild: Charles Petzold: “Programming Windows”, Microsoft Press 1998

Win32-Programmierung: Text-Ausgabe

- **Beispiel Text-Ausgabe** (in **`WndProc`**):

```
   int   iLength;                          /*Anzahl Zeichen*/
   TCHAR szBuffer [40];                   /*String-Speicher*/
   TEXTMETRIC tm;
   static int cxChar,cyChar; /*Vermeidg globaler Var.*/
   /* ... */
case WM_CREATE:    /*Einstellungen einmalig,zu Beginn*/
   hdc = GetDC (hwnd);/*Handle zu Fenster-Daten -s.o.*/
   GetTextMetrics(hdc, &tm);          /*Schrift-Abfrage*/
   cxChar = tm.tmAveCharWidth;      /*count x,y: Pixel*/
   cyChar = tm.tmHeight + tm.tmExternalLeading;
   ReleaseDC (hwnd, hdc);
   return 0;
   /* ... */
case WM_PAINT:           /*Ersatz für DrawText() - s.o.*/
   iLength=sprintf(szBuffer,
              TEXT("HelloWin, %dx mit TextOut!"), 1);
   TextOut (hdc, 2*cxChar, cyChar, szBuffer, iLength);
```

Start: 2 Leerzeichen, 1 Leerzeile

Win32-Programmierung: Text-Eingabe

Regelfall Text-Eingabe von Tastatur:

- Allg. empfohlen: Eingabe, Korrektur, Einfügung v. Zeichen durch Windows; Applikation übernimmt fertige Eingabe.

  Windows-Programme können (müssen nicht) Tastatur-Ereignisse behandeln – sogar bei Texteingabefeldern.

- Beim Drücken oder Lösen einer Taste erzeugt Tastatur sog. **Scancode** (Tasten-Lageerkennung – vgl. Klaviatur);
- Tastaturtreiber erzeugt aus Scancode **ANSI-Code** (landessprachlich angepaßt – ggf. inkl. Tastatur-Umschaltung mit Alt-Shift); er wird von Windows in eine **systeminterne Nachrichten-Warteschlange** gestellt.
- Auf ”Anfrage” der Nachrichten-Warteschleife in der Applikation (z.B.: `GetMessage`) wird Code aus einer systeminternen in die **fenstereigene Nachrichten-Warteschlange** der Applikation kopiert.

⇨ Vorgänge in der Nachrichten-Warteschleife (`WinMain`):

```
while (GetMessage  (&msg, NULL, 0, 0))
{ TranslateMessage (&msg);
  DispatchMessage  (&msg);
} return msg.wParam;
```

(Aus: `winuser.h`)

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

Zu schnelle Eingabe / Prellen wird von Windows „aufgehalten“, bis die Applikation alles verarbeiten kann.

- **`GetMessage`** fordert die nächste Nachricht an, sobald die Applikation die vorherige bearbeitet hat;
- **`TranslateMessage`** stellt sicher, daß die nächste Nachricht tatsächlich diesem Fenster gilt (kein `Alt-Tab`, `Alt-F4`, `Ctrl-Esc` etc.);
- **`DispatchMessage`** besetzt (aktualisiert) daraufhin `msg.hwnd`.

Win32-Programmierung: Text-Eingabe

“el. Kontakt mittl. Reihe links geschlossen bzw. geöffnet”

’A’ bzw. ’a’ bzw. ’á’ bzw. ’α’

Unterscheidung zwischen Tasten (Scancode) und Zeichen (ANSI):

- Auf Tastendruck erzeugt Windows immer Ereignisse: **`WM_KEYDOWN`** oder **`WM_KEYUP`** – bzw. (bei Kombination mit der Alt-Taste) **`WM_SYSKEYDOWN`** oder **`WM_SYSKEYUP`**.
- Die Ereignisse werden dem Callback des aktiven Fensters (`WndProc`) übergeben (über `msg.message`).
- **`WM_KEYDOWN`** oder **`WM_KEYUP`** werden von Windows ignoriert.
- System-Tasten-Ereignisse **`WM_SYSKEYDOWN`** oder **`WM_SYSKEYUP`** sollten unbedingt an Windows weitergereicht werden (über `DefWindowProc`).

Win32-Programmierung: Text-Eingabe

Beispiel: möglicher **WndProc**-Ausschnitt

```
LRESULT CALLBACK WndProc (HWND hwnd,     UINT message,
                          WPARAM wParam, LPARAM lParam)
{ /*(...)*/
  switch (message)       /*Es folgt Code fuer Taste...*/
  { case WM_KEYDOWN:           /*..druecken*/ return 0;
    case WM_KEYUP:             /*.. loesen */ return 0;
    case WM_CHAR:    /*..ANSI-/Unicode-Wert*/ return 0;
    case WM_DEADCHAR: /*..ohne Wagenvorlauf*/ return 0;

    case WM_SYSKEYDOWN:  /*..+Alt entsprechend*/ break;
    case WM_SYSKEYUP:    /*          "         */ break;
    case WM_SYSCHAR:     /*          "         */ break;

    case WM_PAINT:   hdc = BeginPaint(hwnd, &ps);
                     /* ... */
                     EndPaint(hwnd, &ps);     return 0;
  } return DefWindowProc(hwnd,message,wParam,lParam);
}
```

Interessanter als Scancode und ANSI:

- **Virtueller Code**: Er ist
  - sprach- u. layout-unabhängige Kennung aller Tasten (mit – und vor allem:) ohne druckbare Zeichen (erzeugt aus dem Scancode durch Tastaturtreiber und Windows),
  - verbindlich und gleich für alle Länder–z.B. Pfeil-, Funktions- Spezialtasten u.ä.,
  - der Inhalt von **`wParam`** (`msg.wParam` – 32 Bit; früher 16 Bit)

- Der virtuelle Code von Buchstaben und Ziffern ist gleich dem ASCII-Code; er wird aber nicht benutzt, weil meist die ASCII-Zeichennachrichten (s.u.) ausgewertet werden.

Der virtuelle Code ist in **`winuser.h`** festgelegt als **`VK_...`** (**v**irtual **k**ey) – z.B.: **`#define VK_SNAPSHOT 0x2C`**

Dec	*Hex*	*Identifier*	*IBM-Compatible KB*
27	1B	VK_ESCAPE	Esc
33	21	VK_PRIOR	Page Up
34	22	VK_NEXT	Page Down
35	23	VK_END	End
36	24	VK_HOME	Home
37	25	VK_LEFT	Left Arrow
38	26	VK_UP	Up Arrow
39	27	VK_RIGHT	Right Arrow
40	28	VK_DOWN	Down Arrow
44	2C	VK_SNAPSHOT	Print Screen
45	2D	VK_INSERT	Insert
46	2E	VK_DELETE	Delete

Bisher: Tasten-Überprüfung als Einzelereignis betrachtet.

- **`GetKeyState`** liefert Modifier-Zustand bei Tasten-Ereignis – z.B.:

  ```
  short nState = GetKeyState(VK_SHIFT);
  ```

  Setzt das 16. (high) Bit von **`nState`** (neg. Wert), wenn zum Zeitpunkt des momentan bearbeiteten (**!**) Ereignisses die Umschalttaste gedrückt war.

  Entsprechend gibt es: **`VK_CONTROL`** (Ctrl), **`VK_MENU`** (Alt)

- oder auch zur Unterscheidung links / rechts:
  `VK_LSHIFT`, **`VK_RSHIFT`**, **`VK_LCONTROL`**,
  `VK_RCONTROL`, **`VK_LMENU`**, **`VK_RMENU`**.
- und für die Maus:
  `VK_LBUTTON`, **`VK_RBUTTON`**, **`VK_MBUTTON`**.

Win32-Programmierung: Text-Eingabe

Übliche Arbeitsweise:

Überlassung der meisten Nachrichten an Windows, insb. **`WM_SYSKEYDOWN/WM_SYSKEYUP`** (aber auch **`WM_KEYUP`**)

- Arbeit meist mit **`WM_KEYDOWN`**, vor allem für Pfeil-Tasten

  ⇨ Cursor-Steuerung über **`WM_KEYDOWN`** (u.U. mit Shift für Markierung oder Ctrl für wortweise Cursor-Bewegung etc.).

- keine Nutzung bei Schrift:

  Auswertung von **`wParam`** ⇨ **`VK_...`**

  `GetKeyState` ⇨ Shift o.ä. ⇨ länderabhängig – z.B.:

  Shift + ‘3‘ ⇒ '§' (D) oder '#' (US) oder '£' (GB)

  zudem: Anpassungsprobleme mit ANSI- / Unicode-Zeichen

  ⇨ wichtig nur für „nicht-druckbare Aktionen“

Nicht-druckbare Tasten-Kombinationen oft wirkungsgleich mit GUI-Bedienung (Schaltflächen, Bildlaufleisten, Shortcuts, Makros des Benutzers)

⇨ Ineffizient, alles explizit mehrfach zu behandeln: Fehlerquelle, Mehrarbeit bei Änderung o. Bugfixing.

⇨ Lösung: Reaktion auf Nachricht durch neue Nachricht (**`WndProc`** an sich selbst), die das "Original" vortäuscht:

```
SendMessage (hwnd,message,wParam,lParam);
```

`SendMessage`-Parameter wie **`WndProc`**:

`hwnd`: Handle zu bel. Empfänger-Fenster der Nachricht (evtl. fremde App)

`message`: Nachricht-Codierung (**`WM`**_..., Def. in **`winuser.h`**)

`wParam`: 32-Bit-Parameter (Bedeutung ereignisabhängig)

`lParam`: wie **`wParam`** (**`wParam`** früher: 16-Bit-**`WORD`**)

Win32-Programmierung: Text-Eingabe

Beispiel: Beendigung eines Windows-Programms mit **<Esc>** durch Senden der Nachricht **WM_DESTROY** oder **WM_CLOSE**

(nicht: **WM_QUIT**, da direkter **WndProc** -Aufruf):

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,
                         WPARAM wParam,LPARAM lParam)
{ /*...*/
  switch (message)
  { case WM_KEYDOWN:    /*ebenso: case WM_CHAR:*/
        switch (wParam)
        { case VK_ESCAPE:
              SendMessage(hwnd,WM_DESTROY,0,0);
      /*ebenso: SendMessage(hwnd,WM_CLOSE,0,0);*/
              break;
        }
  }
}
```

Win32-Programmierung: Text-Eingabe

Ergeben die gedrückten Tasten ein druckbares Zeichen (z.B.: ‚N' bei Texteingabe, nicht im Dialogfeld mit „J/N"), so erzeugt **`TranslateMessage(&msg)`** aus Tasten- sog. **Zeichennachrichten**.

Für den Wert von **`msg.message`** gilt hierbei:

Nachrichten des Typs:	**erzeugen**	**Nachrichten der Typen:**
`WM_KEYDOWN`		**`WM_CHAR`** (Zeichen)
	oder	**`WM_DEADCHAR`** (Akzente u.ä.)
`WM_SYSKEYDOWN`		**`WM_SYSCHAR`** (Zeichen)
	oder	**`WM_SYSDEADCHAR`** (Akzente)

`msg.wParam` (bei Tastennachrichten belegt mit dem virtuellen Code der Taste) erhält nun den Zeichen-Code.

Win32-Programmierung: Text-Eingabe

Beispiel: Eingabe von ‘A’ ist verbunden mit folgenden Aktionen, Nachrichten (in **`msg.message`**) und Code (in **`msg.wParam`**) :

Aktion	**Nachricht**	**Code**
Umschalttaste drücken	**`WM_KEYDOWN`**	Virtueller Code **`VK_SHIFT`** (0x10)
Taste ‘A‘ drücken	**`WM_KEYDOWN`**	Virtueller Code für 'A' (0x41 - nicht dokumentiert)
_	**`WM_CHAR`**	Zeichen-Code für 'A' (0x41 = 65_{10})
Taste ‘A‘ lösen	**`WM_KEYUP`**	Virtueller Code für 'A' (0x41 - nicht dokumentiert)
Umschalttaste lösen	**`WM_KEYUP`**	Virtueller Code **`VK_SHIFT`** (0x10)

`(KeyView1.exe)`

Fazit für die Text-Eingabe:

- Programme sollten für Textzeichen auf **`WM_CHAR`**, für Cursor-, Funktions- u.ä. Tasten auf **`WM_KEYDOWN`** reagieren.

- **`WM_DEADCHAR`**-Behandlung insg. entbehrlich, da in Windows enthalten – etwa: Umwandlung von <Akzent> + <Konsonant> in zwei Ereignisse, inkl. Korrektur (z.B.: ´s).

- Rücktaste, Enter, Tab, Esc können wahlweise wie Tasten druckbarer (**`WM_CHAR`**) oder nicht-druckbarer (**`WM_KEYDOWN`**) Codes zu behandeln.

Beispiel: Behandlung von Rücktaste, Enter, Tab wie Tasten druckbarer Codes (**`WndProc`**-Ausschnitt):

```
switch (message)
{ case WM_CHAR:
        /* (...) */
      switch (wParam)
      { case '\b':  /*(...)      backspace*/    break;
        case '\t':  /*(...)            tab*/    break;
        case '\n':  /*(...)       linefeed*/    break;
        case '\r':  /*(...)carriage return*/    break;
        default:    /*(...)character codes*/
             sprintf (szBuffer,TEXT("%c"),(TCHAR)wParam);
             InvalidateRect (hwnd, NULL, TRUE);
             break;
      }
      return 0 ;
}
```

Win32-Programmierung: Text-Eingabe

Beispiel: Ausgabe eines Zeichens, Beendigung mit <Esc>:

```
LRESULT CALLBACK WndProc (HWND hwnd,     UINT message,
                          WPARAM wParam, LPARAM lParam)
{ HDC         hdc;              // Geraetekennung
  PAINTSTRUCT ps;               // Art der Zeichenvorgangs
  RECT        rect;             // Zeichenflaeche
  int   iLength=1;              // Anzahl Zeichen
  static TCHAR szBuffer[2];     // Speicher ohne glob.Var.
  TEXTMETRIC tm;
  static int cxChar, cyChar;  //Vermeidg globaler Var.
  switch (message)
  { case WM_CREATE:    /*GetTextMetrics etc.*/ return 0;
    case WM_PAINT:              //Fenster neufuellen:
         hdc=BeginPaint (hwnd, &ps);// Fenster loeschen
         GetClientRect (hwnd,&rect);// Flaeche ermitteln
         TextOut (hdc,2*cxChar,cyChar,szBuffer,iLength);
         EndPaint (hwnd, &ps);      //  Geraet-Freigabe
         return 0;
```

Win32-Programmierung: Text-Eingabe

Beispiel: (Forts.) :

```
/*(...) switch (message) (...)*/
case WM_DESTROY: /*PostQuitMessage etc.*/  return 0;
case WM_KEYDOWN: // ebenso gut: case WM_CHAR:
   switch (wParam)
   { case VK_ESCAPE:    //WM_DESTROY o.WM_CLOSE
     SendMessage (hwnd, WM_DESTROY, 0, 0); break;
   }
 case WM_CHAR:
   switch (wParam)
   { default:                  //(...)character codes
     iLength=sprintf(szBuffer,TEXT("%c"),(TCHAR)wParam);
     InvalidateRect (hwnd, NULL, TRUE);
     break;
   }
  } return DefWindowProc(hwnd,message,wParam,lParam);
}                                                  (HelloWin.exe)
```

Win32-Programmierung: Text-Eingabe

- Die Positionsmarkierung für die Text-Eingabe ('_', '■', v.a. bei Proportionalschrift: '|') heißt unter Windows „**Caret**“. *
- Die wichtigsten Caret-Funktionen betreffen seine Einrichtung bzw. Löschung:
  - **`CreateCaret`** erzeugt ein (unsichbares) Caret für ein Fenster.
  - **`SetCaretPos`** Setzt die Position des Carets im Fenster.
  - **`ShowCaret`** macht das Caret sichtbar (vgl. **`ShowWindow`**).
  - **`HideCaret`** macht das Caret unsichtbar.
  - **`DestroyCaret`** löscht das Caret.
- Weitere Funktionen zur Positions-Abfrage u. für Blinken:
  - **`GetCaretPos`**, **`GetCaretBlinkTime`**, **`SetCaretBlinkTime`**
- Pro Applikation kann nur ein Caret eingerichtet werden; bei mehreren Fenstern muß es alternativ zugeteilt werden
  - **`WM_SETFOCUS`** teilt den Eingabefokus zu (über **`WndProc`**).
  - **`WM_KILLFOCUS`** teilt Fenstern den Entzug des Eingabefokus mit.

Win32-Programmierung: Ressourcen

- **Ressourcen** sind **Daten**, die zu einem Programm gehören, in seiner **.exe**-Datei gespeichert sind, aber nicht beim Start, sondern **nach Bedarf** in den Hauptspeicher geladen werden.

  Dazu gehören: Icons, Cursors, Strings, benutzerdefinierte Daten, Menüs, Keyboard accelerators, Dialoge, Bitmaps.

- Vorteile des Ressourcen-Konzeptes:
  Integration der Daten ins Programm, aber
  - keine Trennung zwischen Programm- u. Daten-Dateien

    ➥ leichtere Installation, ...
  - keine ”Assimilation” als Array

    ➥ Lesen der Icons auch ohne Programm-Ausführung, ...

Bei Ressourcen-Einrichtung werden 2 Text-Dateien angelegt:

- Das **Ressourcenskript `<Programmname>.rc`** mit einem Verzeichnis der verwendeten binären Ressourcendateien
  ⇨ darstellbar grafisch o. als Text (in MS-VC o. im Editor)

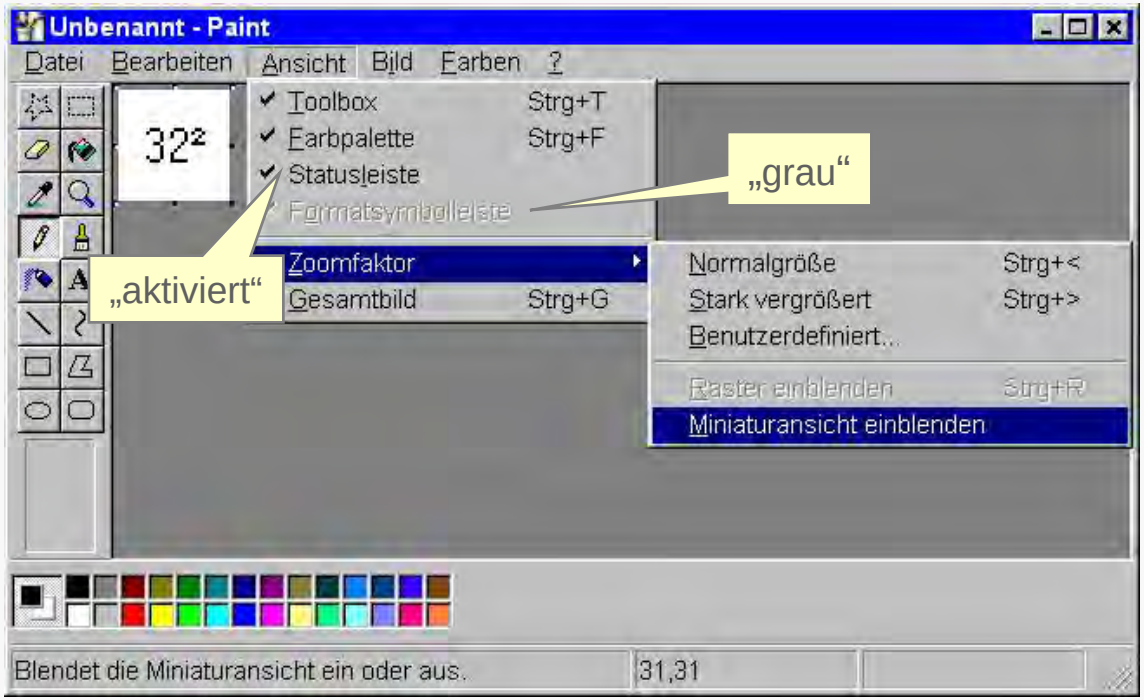
- Die Include-Datei **`resource.h`** mit **`#define`**-Konstanten für den Zugriff auf die eingerichteten Ressourcen

  Der **Ressourcen-Compiler `rc.exe`**
  erzeugt aus Dateien u. Daten in **`*.rc`** eine Datei **`*.res`**;
  diese wird vom Linker in das **`*.exe`** eingebunden.

Menüs als Ressourcen:

- Ein **Menü** ist eine Liste von Wahlmöglichkeiten, jede von denen wiederum Start einer anderen Liste sein kann (Untermenü).
- Das **Systemmenü** wird automatisch jedem Windows-Fenster angehängt.
- Unmittelbar unterhalb der Titelleiste wird die **Menüleiste** (“Hauptmenü”) angelegt. Von den Menüpunkten aus gelangt man nacheinander zu weiteren Menüs (Popup-Menüs als Untermenüs).
- Am Ende eines Menüs / einer Menüfolge steht immer eine **Aktion** oder ein **Dialogfenster**.
- Die **Aktivierung** und die **Wählbarkeit** von Menüpunkten aller Ebenen können miteinander verbunden werden; dabei sind beliebige logische Verknüpfungen realisierbar (Einschluß, Ausschluß, Gruppenbildungen etc.).

- Speziell für Untermenüs gilt:
  - Wahlpunkte, die zu Aktionen führen und gewählt wurden, können als "**aktiviert**" dargestellt werden (mit einem Häkchen links).
  - Nichtwählbare Menüpunkte können "**grau**" dargestellt werden (in hellgrauer Relief-Schrift).



- Menüleisten, Menüs u. Menüpunkte haben eigene **Handles**.

- Drei Chrarkteristika jedes Menüpunktes:

  - **Beschrifttung** und/oder Bebilderung

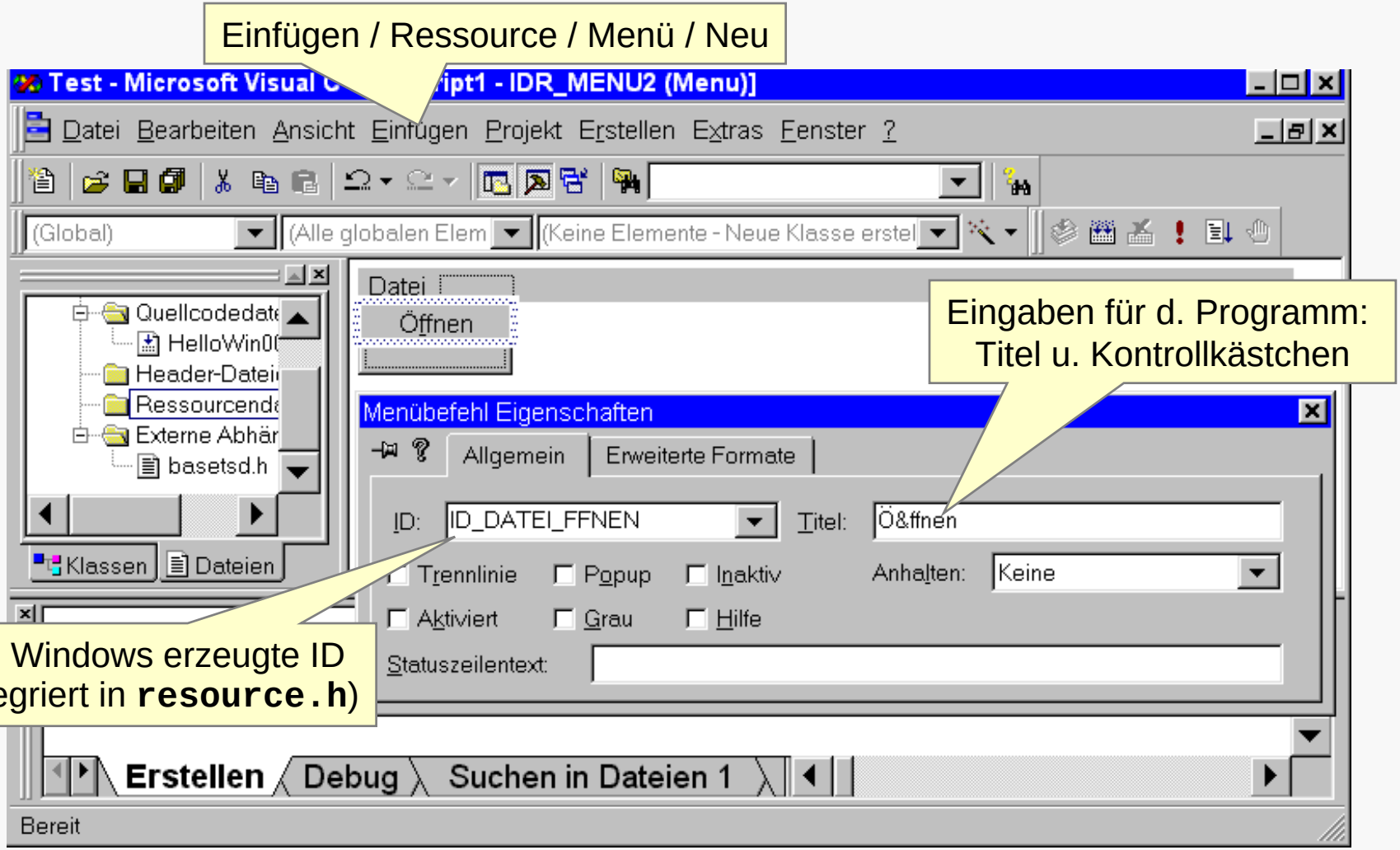
  - **Kennziffer**, mit der Windows der Callback-Funktion die Auswahl dieses Menüpunktes mitteilt (in der Nachricht vom Typ **`WM_COMMAND`**).

  - **Attribute** des Menüpunktes über Art oder Zustand der Aktivierung und deren Kennzeichnung (Häkchen, grau etc.)

Erwähnenswerte Menü-Nachrichten:

- **`WM_INITMENU`**:
  Menü-Aktivierung (noch vor der Menü-Anzeige)
  – z.B. zur rechtzeitigen Veränderung der Menü-Anzeige
- **`WM_MENUSELECT`**:
  Menü-Auswahl (noch vor der Menü-Wahl - vs. ‚choose‘)
  – z.B. zur rechtzeitigen Änderung d. Cursor-Form (Menü/Text)
- **`WM_INITMENUPOPUP`**:
  Auswahl von Menüpunkten mit Untermenüs
  – z.B. zur Überprüfung, ob Inhalt der Zwischenablage eine Aktivierung von “Einfügen” rechtfertigt
- **`WM_COMMAND`**:
  Wahl (Anklicken) eines Menüpunktes; dabei: **`lParam=0`** [bei Steuerelementen: Benachrichtigung per **`WM_COMMAND`** mit **`lParam=`**Handle zum Kind-Fenster ⇨ Unterscheidungskriterium, falls Menüpunkte und Steuerelemente dieselben Kennzahlen haben]

VC-Menü-Editor:



Eingaben der Menübefehl-Eigenschaften:

- **Titel:** Menüpunkt-Beschriftung (mit &*Z*ugriffstaste zur Kombination Alt+*Z* für das Hauptmenü bzw. *Z* für die Menüpunkte)
- **ID:** Aus Titel(-folgen) automatisch erzeugtes Symbol für **`#define`**-Anweisungen in **`resource.h`** (ohne Umlaute u.ä. – nicht für Popup-Menüs)

- **Kontrollkästchen:**

  Popup:

⇨ (aktiviert: Eintrag in **`<Programmname>.rc`**: **`POPUP`**) Popup-Menü – d.h. Startpunkt ab Menüleiste o. im Menü; (deaktiviert: Eintrag **`MENUITEM`**) Menüpunkt

⇨ *Grau*:
(in **`*.rc`**: **`GRAYED`**): Menübefehl zu Beginn inaktiv und grau. (Auswahl erzeugt keine Nachrichten.)

- *Inaktiv*:
  (in **`*.rc`**: **`INACTIVE`**): Menübefehl zu Beginn inaktiv und normal dargestellt. (Auswahl erzeugt keine Nachrichten.)
- *Trennlinie*:
  (in **`*.rc`**: **`SEPARATOR`**): Element zur optischen Gestaltung (kein Befehl)
- *Hilfe*:
  (in **`*.rc`**: **`HELP`**): Menübefehl rechtsbündig auf der Menüleiste angeordnet.
- *Aktiviert*:
  Menübefehl zu Beginn aktiviert
- *Statuszeilentext*:
  nur bei Ressourcendateien mit Unterstützung der Microsoft Foundation Class-Bibliothek (MFC) verfügbar.

Menü einbinden:

(Nach Eingabe von **`#include "resource.h"`**)

- Übergabe der Menü-Kennung an d. Fensterklasse – z.B.:

```
wndclass.lpszMenuName = (LPCSTR)IDR_MENU1;
```

- Berücksichtigung im Callback (**`WndProc`**): Auswertung des niederwertigen Wortes von **`WM_COMMAND`** – z.B.:

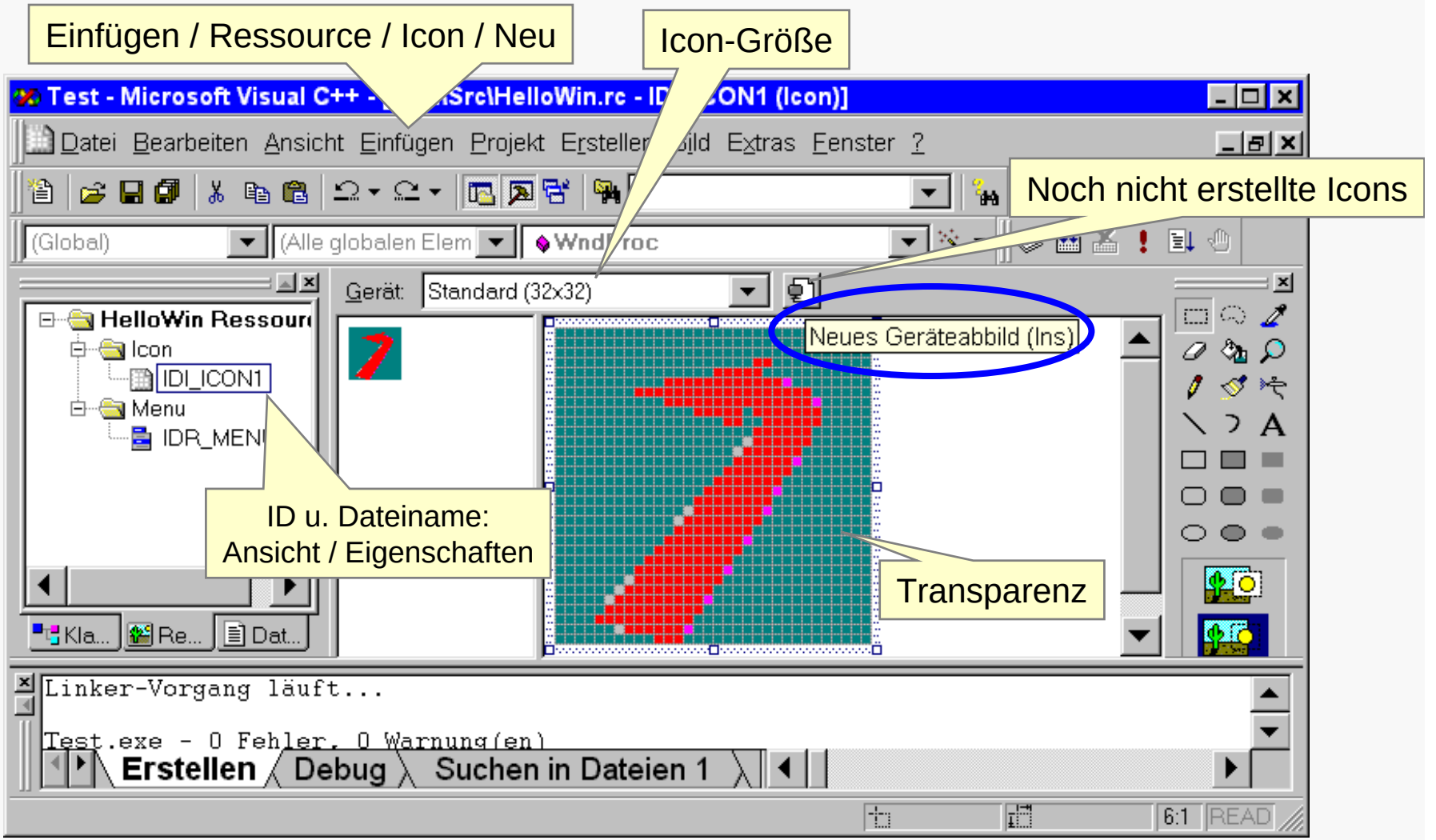
```
switch (message)
{ case WM_COMMAND:
        switch (LOWORD(wParam))
        { case ID_DATEI_BEENDEN:
                SendMessage(hwnd,WM_CLOSE,0,0);
                return 0 ;
          default: MessageBeep (0) ;
                   break;
        }
}
```

Icons als Ressourcen:

- **Icons** begleiten ein Progamm als Kennzeichen seiner ”Individualität”
  - an der Titelleiste des Hauptfensters (o.l.)
  - in Dateilisten des Explorers
  - bei Einträgen im Start-Menü
  - bei Verknüpfungen
  - in der Task-Leiste

- Für Einsätze von großen / kleinen Symbolen werden zwei Icon-Auflösungen unterhalten: 32x32 und 16x16 Pixel.

VC-Icon-Editor:



Icon einbinden:
(Nach Eingabe von **`#include "resource.h"`**)

- Nur Übergabe der Menü-Kennung an die Fensterklasse – z.B.:

```
wndclass.hIcon =
        LoadIcon(hInstance,(LPCSTR)IDI_ICON1);
```

Programm-Handle

Long Pointer To Character String

Icon-ID

Bisher:

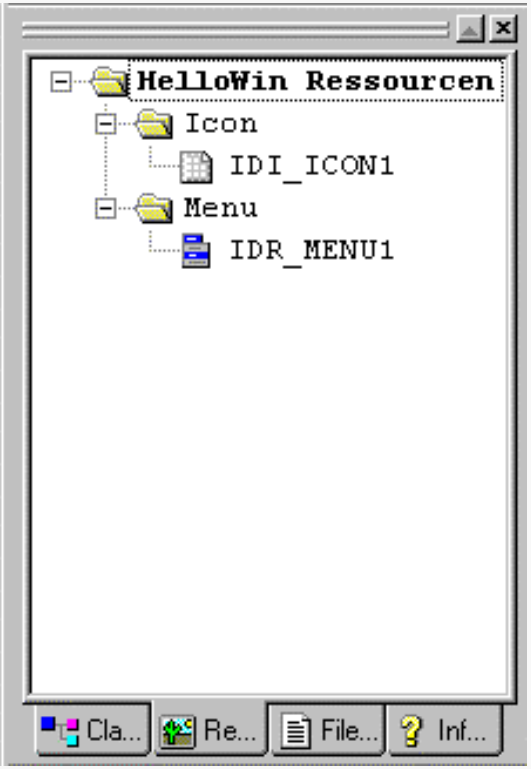
```
wndclass.hIcon =LoadIcon(NULL,IDI_APPLICATION);
```

vordefinierte Icons

- Bei Speicherung mehrerer Auflösungen (32x32, 16x16): korrekte Handhabung durch Windows („große/ kleine Symbole“)
- Weiterentwicklung **`WNDCLASSEX RegisterClassEx`** bietet getrennte Speicherung großer/ kleiner Icons, hat dafür nicht die Flexibilität der automatischen Erkennung großer/ kleiner Icons.

Win32-Programmierung: Ressourcen

ResourceView



HelloWin.rc im Editor (bereinigt):

```
//Microsoft Developer Studio generated
// resource script.
#include "resource.h"


// Menu
IDR_MENU1 MENU DISCARDABLE
BEGIN
  POPUP "&Datei"
  BEGIN
     MENUITEM "Ö&ffnen", ID_DATEI_FFNEN
     MENUITEM "Beenden", ID_DATEI_BEENDEN
  END
END


// Icon
IDI_ICON1 ICON DISCARDABLE "GraphXico.ico"
```

„Bei Platzbedarf freizugeben!“

Win32-Programmierung: Steuerelemente

Steuerelemente (Controls) sind (Child- o.) Kind-Fenster, die

- Maus- und Tastaturereignisse über eine Fenster-Prozedur bearbeiten,
- Reaktionen auf diese Aktionen des Benutzers optisch anzeigen (Direkte Manipulation) und
- das (Eltern- bzw.) Hauptfenster über Veränderungen seines Status benachrichtigen

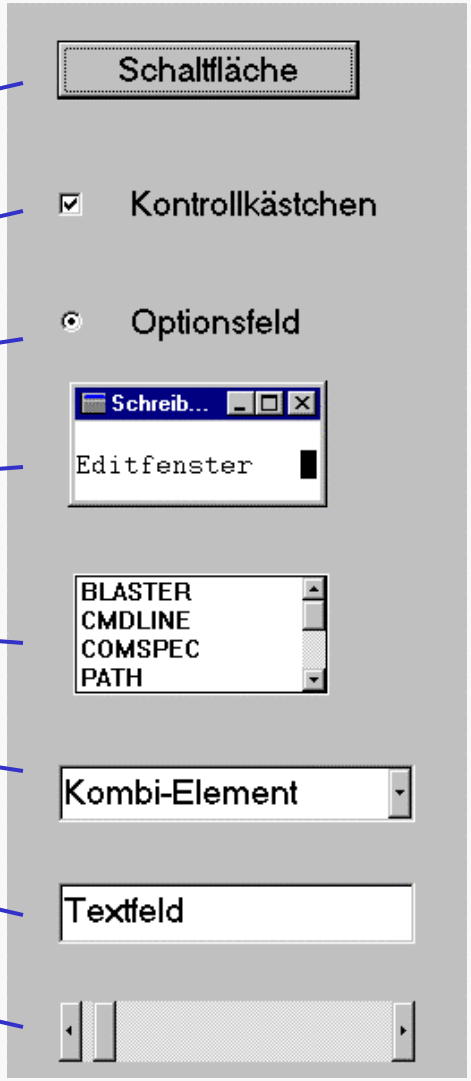
und so die Rolle eines **Eingabemedium**s gegenüber dem Hauptfenster spielen.

Windows übernimmt
- Neuzeichnen (Refresh),
- Trefferprüfung für die Maus,
- Erkennung v. Einfach- o. Doppelklick,
- Eingabefokus für die Tastatur,
- Darstellung diverser Zustände (gedrückt, verschoben, aktiviert etc.).

Win32-Programmierung: Steuerelemente

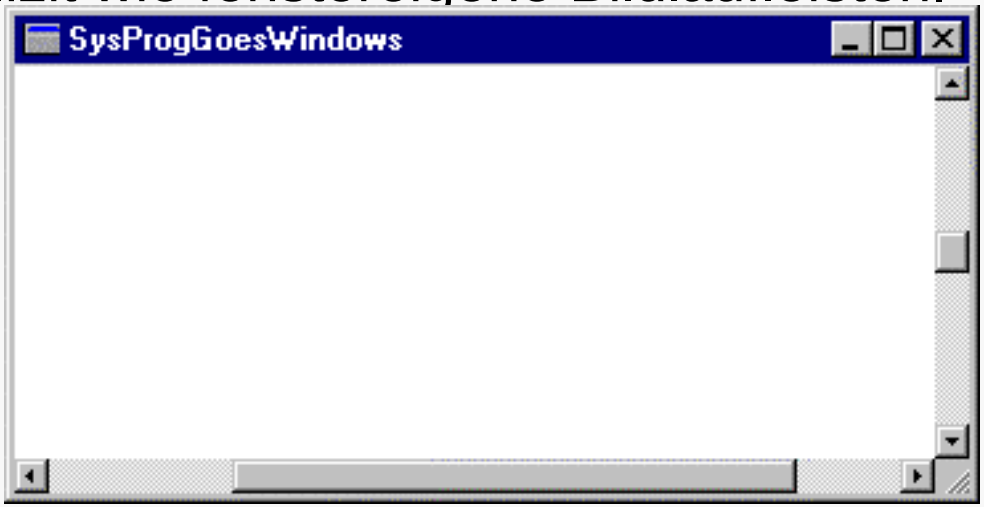
Zu Steuerelementen zählen

- Schaltflächen,
- Kontrollkästchen,
- Optionsfelder,
- Editfenster,
- Listenfenster,
- Kombi-Elemente,
- Textfelder,
- Bildlaufleisten.



Anmerkungen:

- Steuerelemente gehören zur ständigen Ausstattung eines Programms (anders als Ressourcen).
- Die Einrichtung erfolgt in **`WndProc`** als Reaktion auf die Erzeugung des Eltern-Fensters – also auf **`WM_CREATE`**.
- Für häufig vorkommende Anwendungen gibt es vereinfachte Einrichtung – implizit wie die Titelleiste-Schaltflächen o. explizit wie fenstereigene Bildlaufleisten.



Fenstereigene Bildlaufleisten...

- lassen sich nur am rechten und unteren Fensterrand anbringen (⇨ Schaltpult-Darstellungen benötigen die Steuerelement-Version)
- werden eingerichtet, indem im Aufruf **`CreateWindow`** der Fensterstil-Parameter um die entsprechenden Angaben für vertikale und horizontale Bildlaufleisten erweitert (d.h.: ODER-verknüpft) wird:

  `WS_OVERLAPPEDWINDOW` **`| WS_VSCROLL | WS_HSCROLL`**
- legen ihren Wertebereich (z.B. horiz.) fest mit dem Aufruf:
  `SetScrollRange (hwnd, SB_HORZ, 0, 100, FALSE);`
- können (z.B. bei vert. Ausrichtg.) positioniert werden mit:
  `SetScrollPos (hwnd, SB_VERT, 50, TRUE);`

  [hier: 0..100 (def.); letzter Parameter **`TRUE`**, wenn nach dem jew. Aufruf die Bildlaufleiste neuzuzeichnen ist]

Nachrichten für Bildlaufleisten...

- entstehen paarweise bei jedem Mausklick: beim Drücken und beim Lösen;
- werden in **`wParam`** übermittelt:
  - **`HIWORD(wParam)`** enthält die Markenposition (16-Bit-**`int`**),
  - **`LOWORD(wParam)`** zeigt die Aktion an;
- können je nach horizontaler o. vertikaler Ausrichtung sein:
  - **`SB_LINEUP,SB_LINELEFT,SB_PAGEUP,SB_PAGELEFT,`**
  - **`SB_LINEDOWN, SB_LINERIGHT,`**
  - **`SB_PAGEDOWN, SB_PAGERIGHT, SB_ENDSCROLL,`**
  - **`SB_THUMBPOSITION, SB_THUMBTRACK,`**
  - **`SB_TOP, SB_LEFT, SB_BOTTOM, SB_RIGHT,`**

  wobei die **`SB`**_xxxx**`UP`** u. ...**`LEFT`**, **`SB`**_xxxx**`DOWN`** u. ...**`RIGHT`** jeweils denselben numerischen Wert belegen;

Maus-Nachrichten für Bildlaufleisten

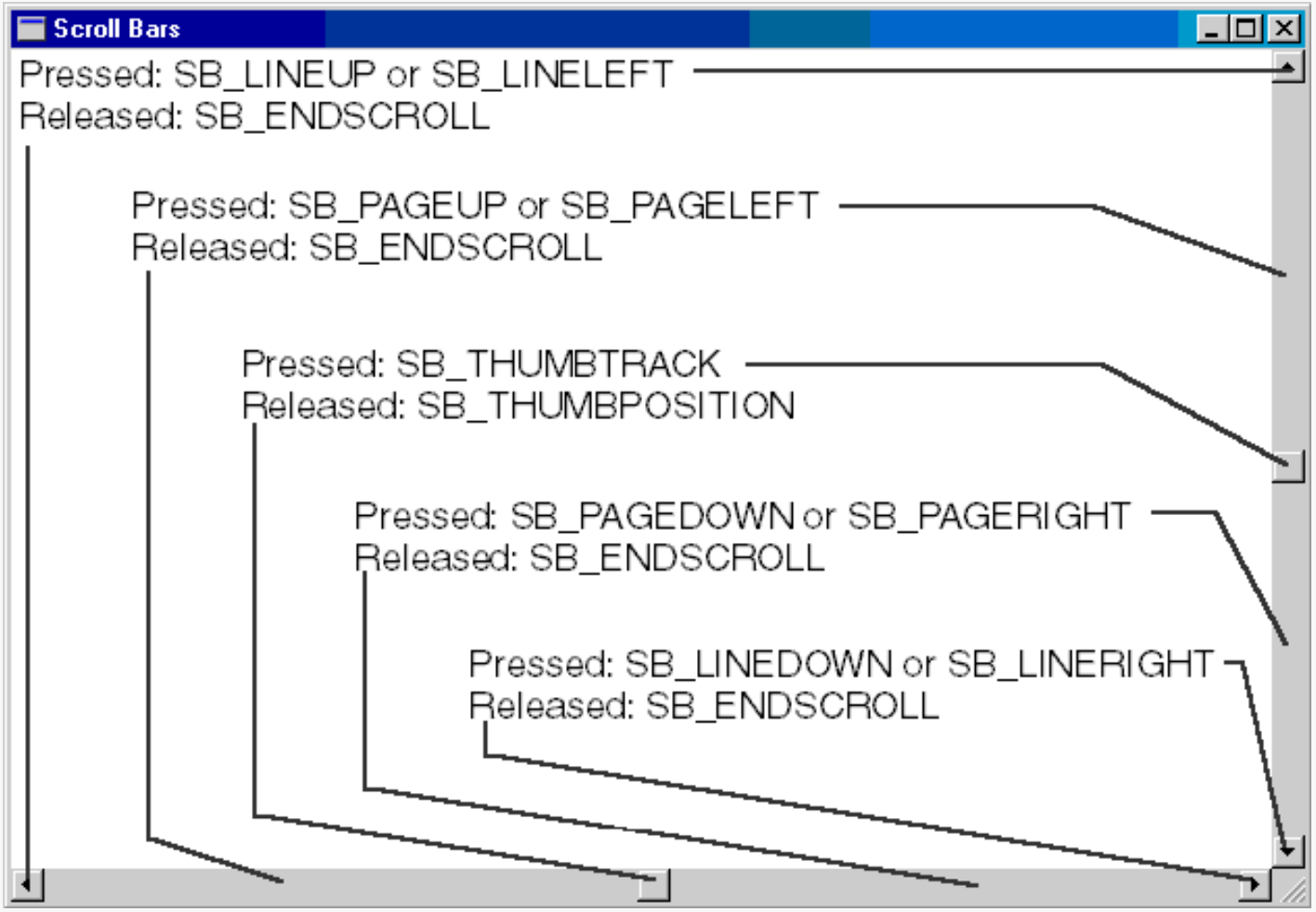


Bild: Charles Petzold: “Programming Windows”, Microsoft Press 1998

- Der Zustand einer Bildlaufleiste ist beschrieben in der Struktur **`SCROLLINFO`** in **`winuser.h`**:

```
typedef struct
{ UINT cbSize;   //set to sizeof(SCROLLINFO)
  UINT fMask;    //bitflags for values to set or get
  int  nMin;     //minimum range value
  int  nMax;     //maximum range value
  UINT nPage;    //page size
  int  nPos;     //current position
  int  nTrackPos;//current tracking position
} SCROLLINFO;
```

- Die Applikation benötigt daher (mind.) die Anweisungen:

```
SCROLLINFO si;
/* Zur Erkennung der zugrundeliegenden API-Version
(für Aufwärtskompatibilität): */
si.cbSize = sizeof(SCROLLINFO);
/* Anweisungen zum Setzen oder Abfragen folgen ...*/
```

si.fMask ist eine bel. bitweise OR-Verknüpfung von Bitflags (Scroll Info Flags SIF), definiert in **winuser.h**:

```
                                        // Lesen o. Setzen von:
#define SIF_RANGE              0x0001 //nMin, nMax
#define SIF_PAGE               0x0002 //nPage:Seitengröße~Markenlänge
#define SIF_POS                0x0004 //nPos: aktuelle Marke-Position
#define SIF_DISABLENOSCROLL    0x0008 //(nur Setzen:) Abgeschaltete
                                      // BL-leiste sichtbar lassen
#define SIF_TRACKPOS           0x0010 //(nur Lesen:) aktuelle Marke-
                                      // Position
#define SIF_ALL       (SIF_RANGE|SIF_PAGE|SIF_POS|SIF_TRACKPOS)
```

- Die Werte von **SCROLLINFO** können gesetzt und abgefragt werden mit den Funktionen:

```
SetScrollInfo (hwnd, iBar, &si, bRedraw);
GetScrollInfo (hwnd, iBar, &si);
```

  mit:

- **hwnd**: Fenster-Handle
- ***iBar:*** **SB_VERT** oder **SB_HORZ** – alternativ: **SB_CTL** (scroll bar control); dann steht der Handle **hwnd** für eine Bildlaufleiste als eigenständiges Steuerelement.
- **bRedraw**: **TRUE**, wenn die Bildlaufleiste nach dem Setzen neuzuzeichnen ist

Beispiel:

```
si.cbSize = sizeof (SCROLLINFO); //Obligatorisch
si.cbMask = SIF_RANGE|SIF_PAGE; //nMin, nMax u. Markenlänge..
si.nMin   = 0;                  //3 „Vorschläge“, werden ggf.
si.nMax   = NUMLINES - 1;       //von SetScrollInfo korrigiert
si.nPage  = cyClient / cyChar;  //Zeilenzahl im Fenster
SetScrollInfo(hwnd,SB_VERT,&si,TRUE);//..setzen, Neuzeichnen
```

`SetScrollInfo` führt z.T. Plausibilitätsüberprüfungen durch.

Es setzt z.B. die (Länge) der Bildlaufmarke nach der Formel:

Markenlänge **:** Leistenlänge =
Größe dargestellter Ausschnitt **:** Gesamtgröße

(alle Größen in Bildschirmpixel)

Zur Programmiertechik:

- Ereignisse, die zum Neuzeichnen führen (Verdeckung, Verschiebung etc.), sollten **`InvalidateRect`** aufrufen u. damit **`WM_PAINT`** auslösen: Vermeidung von Code-Wdhlgn ⇨ bessere Wartbarkeit, weniger Fehlerquellen.

- Maßnahme gegen die niedrige Priorität von **`WM_PAINT`**:

```
InvalidateRect (hwnd, NULL, TRUE);
UpdateWindow (hwnd);
```

Löschen

gesamter Anwendungsbereich

  - **`InvalidateRect`** reiht **`WM_PAINT`** in die Warteschlange ein.
  - **`UpdateWindow`** „sendet“ d. **`WM_PAINT`**-Nachricht an WndProc (direkter vs. indirekter Aufruf der Fenster-Prozedur).

(Verzögertes **`WM_PAINT`** v. **`InvalidateRect`** bleibt wirkungslos.)

Win32-Programmierung: Text-Eingabe (Nachtrag)

Festlegung der Schrift in **`WndProc`** (längster Aufruf: 14 Parameter)

```
HFONT hfont;
DWORD dwCharSet=DEFAULT_CHARSET;
hfont = CreateFont( 20,  // Height (Höhe)
                     0,  // Width (Breite)
                     0,  // Escapement (Schreibrichtg)
                     0,  // Orientation (Neigung)
                     0,  // Weight (Stärke)
                     0,  // Italic (Kursiv)
                     0,  // Underline (Unterstreichung)
                     0,  // StrikeOut (Durchstreichung)
       DEFAULT_CHARSET,  // Charset (Zeichensatz)
                     0,  // OutputPrecision (Exaktheit)
                     0,  // ClipPrecision (Randschnitt)
                     0,  // Quality (Schönschrift)
           FIXED_PITCH,  // Pitch&Family (proportional)
                  NULL   // Font
                       );
SelectObject(hdc, hfont);
```

Übung: (Wdhlg.)

Als Vorarbeit zu einem einfachen Schreibprogramm („Editor“) schreiben Sie eine kurze Windows-Anwendung:

- Sie beginnen mit einem leeren Projekt für eine Win32-Anwendung; darauf bauen Sie wie in der Vorlesung besprochen ein „Hello, Windows!“-Programm auf.
- Sie bereiten das Programm auf die Einrichtung von Ressourcen ein.
- Sie fügen zunächst eine Menüleiste ein, die unter dem Datei-Menü mindestens eine Wahlmöglichkeit zur Programm-Beendigung enthält.
- Sie erstellen ein Icon (32x32 und 16x16 Pixel) und binden es ein.

Übung: (Forts.)

Erstellen Sie ein einfaches Schreibprogramm („Editor“) als Windows-Anwendung mit folgenden Eigenschaften:

- Das Programm startet mit einem 350x300-Fenster, voll geschrieben mit der nicht-proportionalen Version (**`FIXED_PITCH`**) der Standardschrift (**`DEFAULT_CHARSET`**).
- Es kann das Vierfache der zunächst sichtbaren Textgröße aufnehmen (nicht erweiterbar); diese wird sichtbar durch Vergrößern des Fensters oder durch Bedienen einer (immer vorhandenen) vertikalen und einer (bei Bedarf eingesetzten) horizontalen Bildlaufleisten.
- Es arbeitet im Überschreibmodus u. beschreibt zyklisch d. verfügbaren Speicher; Löschen erfolgt über d. Rücktaste.
- Als Ressourcen dienen ein Icon Ihrer Wahl (16x16,32x32) und eine Menüleiste mit dem einzigen Eintrag „Datei“ und den Menüpunkten „Öffnen“ (leeres Fenster anbieten), „u.s.w.“ (nicht aktiv), „Beenden“ (dito). Inaktive Menüpunkte u. Bildlaufleisten-Nachrichten reagieren mit Piepsignal (**`MessageBeep(0);`**)

Übung: (Forts.)

- Verwenden Sie die Fensterklasse **`WNDCLASS`** (NICHT: **`WNDCLASSEX`**) und speichern Sie die Dateien Ihres Projektes wie in **`vcProjekte.doc`** geschildert:
  - **`*.dsp`** , **`*.dsw`** unter **`.\Debug`**,
  - **`*.exe`** unter **`.\Exc`**,
  - **`*.c`** , **`*.h`** , **`*.ico`** , **`*.rc`** unter **`.\Src`**,
  - alle weiteren Dateien unter **`.\tmp`**.
- Versuchen Sie, durch bedingte Kompilierung möglichst getrennte Versionen zu erstellen für Programme
  - mit einer unveränderlichen Ausgabe (z.B.: **`"SysProg Goes Windows"`**) in einem Fenster voreingestellter Größe,
  - mit einem Icon und einem Menü (s.o.), das mindestens die Programm-Beendigung ermöglicht,
  - mit Bildlaufleisten und Text (s.o.) ohne Ressoucen,
  - mit allen besprochenen Merkmalen.
- **Deadline:** 1 Woche nach Klausurtermin **!**