C und C++ für UNIX, DOS und MS-Windows, Teil 2:

MS-Windows-Programmierung

Dies ist weder ein Manual noch ein "normales" Vorlesungs-Skript ("normale" Vorlesungen über eine Programmiersprache sind wohl ohnehin langweilig). Es soll in erster Linie eine Hilfe zum Selbststudium sein (und wird deshalb im folgenden als "Tutorial" bezeichnet).

Die im Skript abgedruckten Programme (Quelltext) können über die Internet-Adresse

http://www.fh-hamburg.de/rzbt/dankert/c_tutor.html

kopiert werden.

Inhalt (Teil 2)

Das waren noch Zeiten, als der Programmierer entschied, wann ein Eingabegerät abgefragt wird. Heute darf jedes Mäuschen ständig ungefragt Botschaften piepsen.

9 Grundlagen der Windows-Programmierung

Windows-Programmierung gilt als schwierig. Bei dieser wohl kaum zu widerlegenden Aussage sollte man unterscheiden zwischen

- der Schwierigkeit, ein völlig neues Konzept verstehen zu müssen, weil kaum jemand mit Erfolg gleich mit Windows-Programmierung anfangen kann, es also unter den Windows-Programmierern wohl ausschließlich "Umsteiger" gibt,
- dem Problem, entweder recht aufwendig immer wiederkehrende ähnliche Programmteile selbst erzeugen, verwalten, anpassen zu müssen, oder aber gleich noch ein zweites neues Konzept zu erlernen und konsequent objektorientiert mit C^{++} zu arbeiten.

Um wirklich effektiv arbeiten zu wollen, muß man wohl sowohl das Windows-Konzept detailliert verstehen als auch objektorientiert arbeiten. Für den Einsteiger in die C-Programmierung, der diesem Tutorial durch die Kapitel 1 bis 8 gefolgt ist, stellt sich die Frage, ob er über das Verständnis der Windows-Programmierung zum objektorientierten Programmieren vordringt oder den umgekehrten Weg geht. Der Streit über den Königsweg ist sicher wieder einen der im Kapitel 1 bereits zitierten "Glaubenskriege" wert.

In diesem Tutorial wird zunächst die Windows-Programmierung mit der Programmiersprache C behandelt, auch deshalb, weil in den Kapiteln 1 bis 8 noch einige spezielle Möglichkeiten nicht erwähnt wurden und "nachgeliefert" werden müssen. Der Vorteil bei dieser Reihenfolge des Erlernens ist sicher, daß das Windows-Programmiermodell deutlich wird. Der Nachteil, immer wiederkehrende Programmteile stets wieder schreiben zu müssen, kann durch geschicktes Arbeiten mit dem Editor weitgehend entschärft werden (und für den Lernenden ist es durchaus nicht nachteilig, diese Programmteile immer noch einmal zu sehen).

Windows-Programmierung ist leider immer noch eine system-spezifische Angelegenheit. Hier wird auf MS-Windows 3.1 aufgebaut. Die Hoffnung, dafür geschriebene Programme nach Neu-Compilierung z. B. auf einem X11-System unter UNIX ablaufen lassen zu können, ist illusorisch (das mag denjenigen enttäuschen, der die ANSI-C-Programme aus den Kapiteln 1 bis 7 z. B. sowohl als "QuickWin-Application" mit MS-Visual-C erzeugt als auch unter UNIX in einem X11-Terminal-Fenster gearbeitet hat, ohne auch nur eine Programmzeile ändern zu müssen). Die nachfolgenden Beispiel-Programme wurden sämtlich mit MS-Visual-C^{++} 1.5 (unter MS-Windows 3.11) getestet, ein Entwicklungssystem, daß alle Strategien der C- und C^{++}-Programmierung unterstützt und deshalb weiter genutzt werden kann, wenn man von der Windows-C-Programmierung auf objektorientiertes Arbeiten umsteigt.

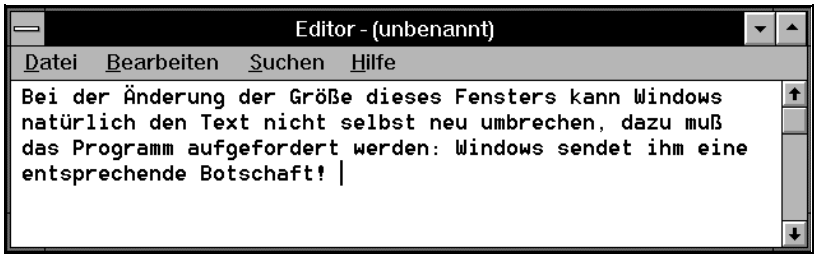
9.1 Das MS-Windows-Konzept

MS-Windows 3.1 wird (wie ein "gewöhnliches" Anwendungsprogramm) unter MS-DOS gestartet, zeigt sich dem Benutzer dann jedoch wie ein selbständiges Betriebssystem. Tatsächlich liegt die Verantwortung für das Dateisystem weiterhin bei den entsprechenden DOS-Routinen, Windows "übernimmt den Rest" (Speicher- und Programmverwaltung und die Steuerung aller Ein- und Ausgabegeräte). Die auffälligsten Besonderheiten des Arbeitens unter MS-Windows (im Vergleich zum Arbeiten unter DOS) sind, daß mehrere Anwendungen gleichzeitig aktiv sind, die sich alle Ressourcen teilen müssen. Genau eine Anwendung hat den **Eingabefokus**, kann also über die Maus und die Tastatur angesprochen werden.

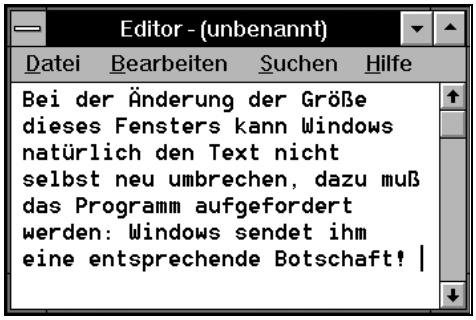
Damit scheint für den Programmierer die Angelegenheit nicht nennswert anders zu sein als bei der "klassischen" Programmierung: An die Stelle des Bildschirms als Ausgabegerät tritt ein Fenster, und mit der Eingabe muß gegebenenfalls gewartet werden, bis das Fenster den Eingabefokus hat. Doch dieser erste Anschein trügt, was deutlich wird, wenn man analysiert, welche Aktionen mit den Fenstern von Windows ausgeführt werden können:

- Das Verschieben eines Fensters (samt Inhalt) auf dem Bildschirm darf man einem Windows-System ohne weiteres zutrauen, weil es ohnehin den Inhalt des Bildschirmspeichers kontrolliert. Die Beantwortung der Frage, ob dabei auch die vorher verdeckten Darstellungen in anderen Fenstern "repariert" werden können, hängt davon ab, ob der Inhalt aller Bereiche vor dem Überzeichnen gespeichert wurde (diese denkbare Variante wird von MS-Windows nicht verfolgt, was den zukünftigen Windows-Programmierer ahnen läßt, was auf ihn zukommt).

- So richtig ins Grübeln müßte der mit den "klassischen Strategien" vertraute Programmierer allerdings bei folgendem Beispiel kommen: Man startet aus dem "Zubehör" von Windows das Programm "Editor" und füllt das Fenster mit einem beliebigen Text. Wenn unter dem Menüpunkt "Bearbeiten" die Option "Zeilenumbruch" gewählt wird, paßt das Programm den Text in das Fenster ein (Bild oben). Wenn man nun die Größe des Fensters ändert (Bild rechts), wird der Text "neu umgebrochen", was natürlich nicht von Windows erledigt werden kann, denn den Text verwaltet ja nur das Anwendungsprogramm "Editor".



Text nach der Wahl der Option "Zeilenumbruch"



Darstellung des Textes nach der Verkleinerung des Fensters

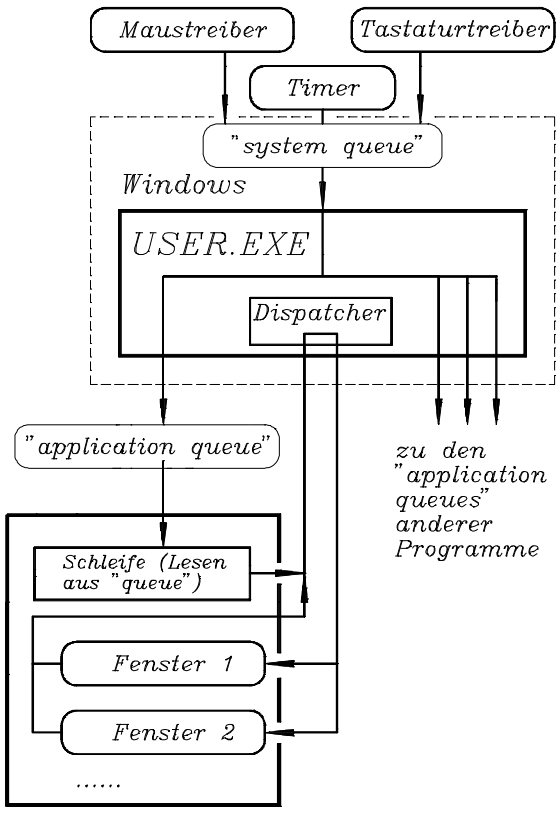
Wie kann ein Anwendungsprogramm erfahren, daß sich die Fenstergröße geändert hat? Schließlich kann es diese nicht ständig beim Betriebssystem erfragen. Hier äußert sich eine ganz neue Strategie, die den

Windows-Programmierer zu einer völlig neuen Denkweise zwingt: Sein Programm kann nicht nur Ereignisse (z. B. eine Eingabeaktion) beim Betriebssystem abfragen, es muß auch auf **Botschaften** reagieren können, die ihm von Windows "gesendet werden". Die Arbeitsteilung bei dem beschriebenen Beispiel sieht also so aus: Windows zeichnet den geänderten Fensterrahmen (einschließlich Titelleiste, Menü, "Schaltflächen" und "Fahrstuhl" usw.) und sendet dem Programm die Botschaft, daß es sich um den Inhalt des Fensters kümmern muß.

9.2 Botschaften (Nachrichten, "Messages")

Natürlich kann ein Programm unter Windows (schon wegen der in anderen Fenstern gleichzeitig aktiven Programme) nicht die Selbständigkeit haben wie ein DOS-Programm. Alle Anwendungsprogramme laufen unter der gemeinsamen Steuerung des Windows-Programms **USER.EXE**. Dieses entnimmt aus einer von Maus- und Tastaturtreiber (und dem Timer) gespeisten Nachrichtenschlange ("system queue") die Botschaften und ordnet sie den einzelnen Applikationsprogrammen zu (die Begriffe "Nachrichten", "Botschaften", "Messages" werden gleichwertig verwendet). Das Programm **USER.EXE** ist also der Verteiler der Botschaften.

Die eigentliche Besonderheit besteht allerdings darin, daß Botschaften nicht nur auf Warteschlangen für die einzelnen Programme ("application queues") verteilt werden (das geschieht allerdings auch), **sondern direkt einem zu einem Programm gehörenden Fenster gesendet werden können**. Da jedes Programm mehrere Fenster öffnen kann, muß für jedes Fenster ein Empfänger der Botschaften definiert werden, eine **Fenster-Funktion**.



Die Skizze verdeutlicht, daß die **Fenster-Funktion (als sogenannte "call back function") niemals direkt aus dem Anwenderprogramm, sondern immer über Windows aufgerufen wird**. Man sieht auch, daß dies nicht nur "als Umweg" bei der Abarbeitung einer aus der "application queue" entnommenen Botschaft geschieht, sondern auch bei Botschaften aus dem Anwendungsprogramm der ganz normale (und einzig mögliche) "Dienstweg" ist, der eingehalten werden muß.

Diese bemerkenswerte Tatsache ist der wohl markanteste Unterschied zur "klassischen Programmierung":

In einem Windows-Programm gibt es mindestens eine Funktion, die nicht von einer anderen Funktion des Programms direkt aufgerufen wird. Einer solchen "Fenster-Funktion" werden von Windows "Botschaften" übergeben.

Selbst dann, wenn eine Funktion des Anwendungsprogramms eine Botschaft an eine Fenster-Funktion senden möchte, muß sie eine Windows-Funktion aufrufen, die ihrerseits dann die Botschaft an die Fenster-Funktion schickt.

Es gäbe noch sehr viel zu erläutern, was eigentlich vor dem ersten Beispiel-Programm geklärt werden müßte, aber es ist sicher eine gute Idee, wie in den ersten Kapiteln dieses Tutorials einfach ein Programm anzugeben, das die typische Struktur eines Windows-Programms verdeutlicht.

9.3 Das kleinste Windows-Programm "miniwin.c"

Mindestens drei Files sind die Quellen für ein Windows-Programm, die C-Quellcode-Datei, die Windows-Header-Datei und eine Definitions-Datei für den Linker:

- Die C-Quellcode-Datei enthält das Hauptprogramm, das **WinMain** heißen muß, und die Fenster-Funktion für das Hauptfenster, die einen beliebigen Namen haben darf (natürlich könnten WinMain und die Fenster-Funktion auch in zwei Dateien untergebracht werden).
- Jede Funktion, die Windows-Funktionen aufruft oder Windows-Datentypen bzw. Windows-Konstanten enthält, muß die (über 5000 Zeilen lange) Header-Datei **windows.h** einbinden, die die Prototypen aller Windows-Funktionen und die Definitionen zahlreicher Datentypen und Konstanten enthält.
- Für den Linker wird eine sogenannte "Definitios-Datei" (Extension **.def**) benötigt, die neben einigen Optionen für den Link-Aufruf die Namen der Fenster-Funktionen enthält. Da die Fenster-Funktionen nicht aus den vom Programmierer geschriebenen Funktionen gerufen werden, wird der Linker auf diese Weise angewiesen, sie trotzdem in das ausführbare Programm mit einzubinden.

Obwohl das nachfolgende Programm ausführlich kommentiert ist, bleiben sicherlich noch viele Fragen offen (die Beschreibung einiger Parameter wurde ohnehin zurückgestellt). Der Einsteiger sollte sich davon nicht abschrecken lassen.

Eine abschreckende Wirkung mag auch der Umfang des Programms haben. Daran sind natürlich die Kommentare wesentlich beteiligt. Im nachfolgenden Abschnitt wird das Programm-Skelett unkommentiert (mit einigen unwesentlichen Änderungen) noch einmal aufgelistet. Aber auch unkommentiert ist **miniwin.c** doch wesentlich umfangreicher als **minimain.c** aus dem Abschnitt 3.3. Immerhin kann man mit der Windows-Version des Mini-Programms Fenster-Operationen ausführen. Die Struktur des Programms kann als Muster für alle weiteren Windows-Programme dienen.

- In Windows-Programmen werden viele Objekte (z. B. auch die Fenster) durch sogenannte **Handles** identifiziert. Dahinter verbergen sich schlichte vorzeichenlose ganze Zahlen.
- Und weil selbst bei dem Mini-Programm der Überblick nur schwer zu gewinnen ist, zunächst die "Kurzform des Minis":
  - **WinMain** legt die Parameter für sein Hauptfenster in einer Struktur vom Typ **WNDCLASS** ("window class") ab und läßt diese Klasse mit **RegisterClass** registrieren, ...
  - kreiert ein Fenster dieser Klasse mit **CreateWindow**, ...
  - das von **ShowWindow** auf den Bildschirm gebracht wird.
  - Danach liest **WinMain** in einer Schleife mit **GetMessage** die von **USER.EXE** in der "application queue" abgelegten Botschaften ...
  - und leitet sie via **DispatchMessage** an die eigene Fenster-Funktion **WndProc_pd**.
  - Die Fenster-Funktion **WndProc_pd** empfängt Botschaften, ...
  - sucht sich diejenigen aus, auf die reagiert werden soll (in diesem Fall wird nur auf die Botschaft **WM_DESTROY** - Schließen des Fensters - reagiert) ...
  - und leitet alle nicht selbst bearbeiteten Botschaften an die für die Standard-Behandlung zuständige Windows-Funktion **DefWindowProc** weiter.

```
/*  Windows-Minimal-Programm
    ========================

    Dieses Skelettprogramm demonstriert den prinzipiellen Aufbau eines
    Windows-Hauptprogramms und einer "Fenster-Funktion".

    In der Windows-Programmierung haben sich einige typische Namen fuer die
    Variablen durchgesetzt, was die Lesbarkeit von Windows-Programmen
    wesentlich erleichtert. An diese Konventionen halten sich auch die
    Programme des Tutorials, in diesem ersten Programm allerdings mit
    folgender Einschraenkung:

    Um zu verdeutlichen, welche Bezeichnungen fest vorgegeben sind (z. B.
    'WinMain' fuer das Hauptprogramm) und welche der Programmierer frei
    waehlen darf (z. B. 'WndProc' fuer die Fenster-Funktion), werden die frei
    waehlbaren durch den Zusatz '_pd' (vom Programmierer definiert) gekenn-
    zeichnet (also z. B.: 'WndProc_pd'). Dieser Zusatz wird in den nachfol-
    genden Programmen weggelassen.                                          */

#include <windows.h>   /* ... muss unter Windows immer eingebunden werden,
                          ist eine sehr umfangreiche Datei, enthaelt neben
                          den Prototypen der fuer die Programmierung verfueg-
                          baren Windows-Funktionen auch die vielen Typ-
                          Definitionen, die nachfolgend verwendet werden.   */


LONG FAR PASCAL WndProc_pd (HWND , UINT , UINT , LONG) ;
                       /* ... ist der Prototyp der unten definierten Fenster-
                          Funktion. Die Typen sind in windows.h definiert:
                          LONG ist "signed long", UINT ist "unsigned int",
                          HWND ("Handle of Window") ist wie alle Handles
                          ebenfalls "unsigned int".
```

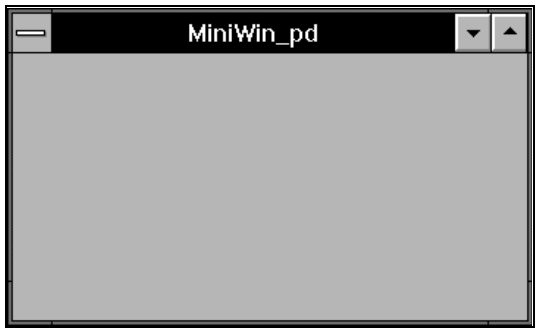
```
                        Die Angabe "PASCAL" ist erforderlich, um das Ablegen
                        der Parameter auf dem Stack bei Funktionsaufrufen
                        abweichend von der in C ueblichen Reihenfolge zu
                        erzwingen, "FAR" haengt mit der leidigen Segmentie-
                        rung unter DOS zusammen und besagt, dass die
                        Funktion in einem anderen Code-Segment als das
                        aufrufende Programm liegen kann. Der Anfaenger
                        sollte sich nur merken, dass alle Fenster-Funktionen
                        prinzipiell als "FAR PASCAL" deklariert werden
                        muessen.                                          */

int PASCAL WinMain (HANDLE hInstance_pd     ,  /* ... Handle auf diese gerade
                                               gestartete "Programm-Instanz".  */
                    HANDLE hPrevInstance_pd ,  /* ... Handle auf die letzte
                                  Vorgaenger-Instanz (NULL, wenn es keine
                                  Vorgaenger-Instanz gibt).              */
                    LPSTR  lpszCmdParam , /* ... pointert auf einen String mit
                                  Kommandozeilen-Parametern, erfuellt die
                                  gleiche Aufgabe wie in einer "C-main"-
                                  Funktion argc und argv gemeinsam.      */
                    int    nCmdShow)  /* ... ist der Fenstertyp (Vollbild,
                                  Icon), mit dem sich das Programm
                                  melden soll.                           */
{
  MSG  msg_pd ;   /* Der Typ "MSG" ist eine Struktur, die eine Botschaft
                     enthaelt. Diese besonders wichtige Struktur wird in
                     windows.h folgendermassen definiert:

    typedef struct tagMSG
          {
                HWND    hwnd    ;    **  ... Handle des Fensters, fuer das
                                          die Botschaft bestimmt ist.      **
                UINT    message ;    **  ... die Botschaft selbst. codiert
                                          als Kennziffer, die in windows.h
                                          definiert wird, z. B. WM_KEYDOWN
                                          fuer das Druecken einer Taste.   **
                WPARAM  wParam  ;
                LPARAM  lParam  ;    **  ... sind zusaetzliche Informationen
                                          zu der Botschaft (WPARAM -->
                                          "unsigned int", LPARAM --> "signed
                                          long"), geben z. B. an, welche
                                          Taste gedrueckt wurde.           **
                DWORD   time    ;    **  ... Zeit, zu der die Botschaft
                                          erzeugt wurde.                   **
                POINT   pt      ;    **  ... Cursor-Koordinaten zu diesem
                                          Zeitpunkt                        **
          } MSG;                                                          */

  HWND hwnd_pd ;          /* ... "Handle of Window HWND" ist auch ein "HANDLE",
                             hier fuer das Hauptfenster des Programms.      */
  WNDCLASS wndclass_pd ; /* ... Window-Klasse (fuer das Hauptfenster)       */

  if (!hPrevInstance_pd)
    {                     /* ... gibt es noch keine registrierte "Fenster-   */
                          /* klasse" fuer dieses Programm, weil es keine
                             Vorgaenger-Instanz des Programms gibt.          */

      wndclass_pd.style         = CS_HREDRAW | CS_VREDRAW  ;    /* ... ist
                     eine typische Kombination von Eigenschaften, die in
                     Parametern bitweise verschluesselt sind und deshalb mit
                     dem "Logischen OR" miteinander verknuepft werden koennen,
                     hier: Fenster ist komplett neu zu zeichnen, wenn sich
                     horizontale oder vertikale Groesse aendert             */

      wndclass_pd.lpfnWndProc   = WndProc_pd ; /* ... damit Windows weiss, wer
                     das eigentliche "Arbeitstier" dieses Programms ist, zum
                     Typ dieser Variablen siehe Kommentar am Programmende   */
      wndclass_pd.cbClsExtra    = 0 ;
```

```
      wndclass_pd.cbWndExtra    = 0 ;
      wndclass_pd.hInstance     = hInstance_pd   ;
      wndclass_pd.hIcon         = LoadIcon (NULL , IDI_APPLICATION) ;
                  /* ... ein vordefiniertes Icon als "Verkleinerungssymbol" */
      wndclass_pd.hCursor       = LoadCursor (NULL , IDC_ARROW)  ;
                  /* ... ein vordefinierter Cursor (Pfeil)                  */
      wndclass_pd.hbrBackground = GetStockObject (LTGRAY_BRUSH)  ;
                  /* ... ein vordefinierter Hintergrund ("Light Gray") wird
                         von GetStockObject "aus dem Lager geholt".         */
      wndclass_pd.lpszMenuName  = NULL                ;
      wndclass_pd.lpszClassName = "WndClassName_pd" ; /* ... gibt der zu
                     registrierenden Fensterklasse einen Namen.             */

      RegisterClass (&wndclass_pd) ;     /* ... registriert die gerade
                                            definierte"Fensterklasse"       */
    }

  hwnd_pd = CreateWindow ("WndClassName_pd" ,   /* ... Fensterklassen-Name  */
                          "MiniWin_pd"      ,   /* ... fuer die Titelleiste */
                          WS_OVERLAPPEDWINDOW , /* ... Fensterstil          */
                          CW_USEDEFAULT     ,   /* ... Position ...         */
                          CW_USEDEFAULT     ,   /* ... und      ...         */
                          CW_USEDEFAULT     ,   /* ... Fenster-             */
                          CW_USEDEFAULT     ,   /* ... groesse              */
                          NULL              ,   /* ... "Parent-Window"      */
                          NULL              ,   /* ... Handle fuer Menue    */
                          hInstance_pd      ,   /* ... Programm-Instanz     */
                          NULL)             ;   /* ... "special parameter"  */

            /* ... kreiert ein Fenster (ohne es bereits anzuzeigen) auf der
                   Basis der registrierten Fensterklasse "WndClassName_pd"  */

  ShowWindow   (hwnd_pd , nCmdShow) ;  /* ... zeigt schliesslich das
                  kreierte Fenster, identifiziert durch "HANDLE hwnd_pd",
                  der Typ "nCmdShow" (Vollbild, Symbol, ...) wird WinMain
                  von Windows mitgeteilt (siehe oben).                      */

  /* Es folgt die Hauptschleife des Programms, die solange durchlaufen wird,
     bis Windows die Meldung WM_QUIT, erkennbar durch den GetMessage-Return-
     Wert 0, liefert:                                                       */

  while (GetMessage (&msg_pd , NULL , 0 , 0))  /* ... holt Botschaft ab ... */
    {
      DispatchMessage  (&msg_pd) ;            /* ... und leitet sie ueber den
                        "Windows-Dienstweg" schliesslich an die "Fenster-
                        funktion" (in diesem Fall WndProc_pd) weiter.       */
    }
       /* ... und diese Schleife laeuft von "Botschaft zu Botschaft", weil
              GetMessage einen Return-Wert gleich 0 nur fuer eine einzige
              Botschaft liefert: WM-QUIT ("Programm beenden").              */

  return msg_pd.wParam ;
}


/**********  Fenster-Funktion, die dem Hauptfenster des Programms  **********/
/**********  zugeordnet wurde:                                     **********/

LONG FAR PASCAL WndProc_pd (HWND hwnd   , UINT message ,
                            UINT wParam , LONG lParam)
                            /* ... erhaelt eine Botschaft, die durch die
                               ersten vier Parameter der oben erlaeuterten
                               "MSG-Struktur" beschrieben wird.             */

{
  switch (message)          /* Von der Vielzahl der Botschaften, die in
                               windows.h definiert sind, kann sich der
                               Programmierer diejenigen aussuchen, auf die
```

```
                                das Programm reagieren soll, hier soll es
                                nur eine sein:                                  */
  {

    case WM_DESTROY:        /* ... "Fenster schliessen" (z. B. durch
                                   Mausklick "links oben") soll ...             */
      PostQuitMessage (0) ; /* ... Nachricht WM_QUIT ausloesen, die in
                                   WinMain das Programmende einleitet.          */
      return 0            ;
  }

  return DefWindowProc (hwnd , message , wParam , lParam) ;
         /* ... und alle Nachrichten, fuer die der Programmierer keine Aktion
                vorgesehen hat, werden an die Windows-Funktion DefWindowProc
                zur "Default-Behandlung" uebergeben.                            */
}


/*  Mit der Anweisung in WinMain

                        wndclass_pd.lpfnWndProc = WndProc_pd ;

    wird der Struktur-Variablen ein "Pointer auf eine Funktion" zugewiesen.
    Ein solcher Pointer wird in C mit der gleichen Symbolik wie ein Pointer
    auf eine Variable gehandhabt. Der Name der Funktion repraesentiert die
    Adresse, ab der der Code der Funktion gespeichert ist, so dass mit der
    Kenntnis dieser Adresse die Funktion aufgerufen werden kann.

    Hier wird also Windows in Kenntnis gesetzt, welche Funktion als "Fenster-
    Funktion" fuer die definierte Fensterklasse eingesetzt werden soll, die
    dann von Windows mit der Dereferenzierung des Pointers aufgerufen wird. */
```

Mit MS-Visual-C^{++} kann folgende Strategie für das Erzeugen des ausführbaren Programms empfohlen werden:

- Nach dem Start der integrierten Entwicklungsumgebung unter Windows werden das Menü-Angebot **Project** und die Option **New** gewählt.
- In dem danach geöffneten Fenster werden als **Project Name** z. B. **miniwin** eingetragen, das Standard-Angebot **Windows application (.EXE)** kann angenommen werden. Nach dem Schließen dieses Fensters ...
- ... wird automatisch ein Edit-Fenster geöffnet, das für die Zuordnung der Files zu dem Projekt genutzt wird. Die Files **miniwin.c** und **miniwin.def** werden ausgewählt und mit der Option **Add** in das Projekt eingefügt. Nach dem Klicken auf die Schaltfläche **Close** ist das Projekt definiert.
- Unter dem Menüangebot **Project** wählt man **Build MINIWIN.EXE**. Das entstehende ausführbare Programm kann ebenfalls unter dem Menüangebot **Project** mit der Option **Execute MINIWIN.EXE** gestartet werden. Es präsentiert sich mit dem nebenstehend abgebildeten Fenster.



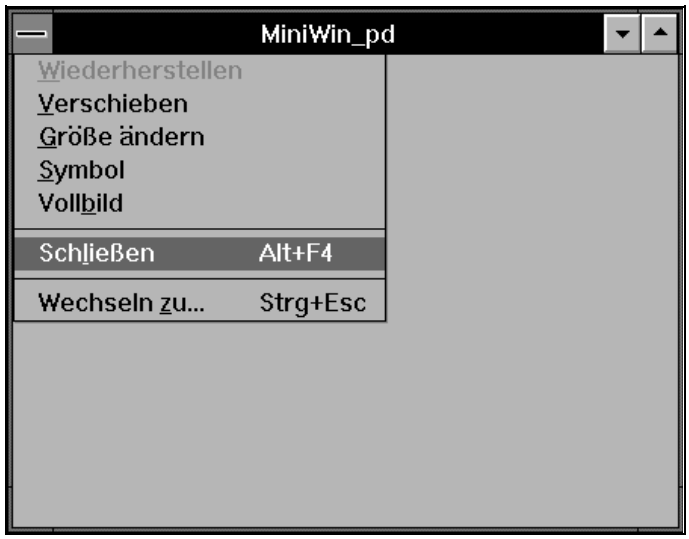
Fenster des Programms **miniwin.c**

Nachfolgend wird eine Definitions-Datei **miniwin.def** angegeben, die der Linker benötigt und die zwingend zum Projekt gehören muß. Sie könnte z. B. folgendermaßen aussehen:

```
; Die Definitionsdatei wird fuer den Linker benoetigt.

; Der NAME (hier: MINIWIN) und die Angaben nach dem Schluesselwort
; EXPORTS muessen dem aktuellen Problem angepasst werden, alle uebrigen
; Zeilen koennen fuer die meisten Programme ungeaendert uebernommen
; werden

NAME      MINIWIN                     ; Das Schluesselwort "NAME" legt fest,
                                      ; dass ein ausfuehrbares Programm (und
                                      ; keine LIBRARY) erzeugt werden soll

; Die folgenden Zeilen entsprechen den Default-Vorgaben von MS-Visual-C++ 1.5
; und sind sinnvoll fuer die meisten "normalen" Programme:

EXETYPE   WINDOWS
CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD MOVEABLE MULTIPLE
HEAPSIZE  1024

; Nach dem Schluesselwort EXPORTS muessen alle Funktionen aufgelistet
; werden, die von Windows direkt angesprochen werden (Ausnahme: WinMain
; braucht nicht aufgefuehrt zu werden), weil diese Funktionen mit selbst
; zu waehlenden Namen versehen werden koennen und im eigentlichen Programm
; nicht aufgerufen werden, so dass der Linker gar nicht wissen kann, dass
; sie gebraucht werden.
;
; In diesem Fall ist es nur die "Fensterfunktion".

EXPORTS
 WndProc_pd @1
```

Definitions-Datei miniwin.def

Die Fensterfunktion des Programms **miniwin.c** reagiert auf keine Botschaft (außer "Fenster schließen"). Trotzdem kann man alle Manipulationen mit dem Programm-Fenster ausführen, weil diese Funktionalität von Windows bereitgestellt wird:



Programm **miniwin.c** mit Menü zur Fenster-Manipulation

- Verschieben durch "Klicken und Ziehen" in der Kopfleiste,
- Fenstergröße ändern durch "Klicken und Ziehen" an Rändern und Ecken,
- Vergrößern auf Bildschirmgröße und Verkleinern zum "Icon" durch Anklicken der Schaltflächen in der rechten oberen Fensterecke (und die Möglichkeiten, diese Aktionen rückgängig zu machen),
- Öffnen eines Menüs durch Anklicken der Schaltfläche in der linken oberen Ecke (Abbildung) zum Verschieben, Verkleinern, Vergrößern des Fensters mit den Cursortasten und zum Schließen des Fensters.

9.4 Windows-Skelett-Programm "winskel.c"

Das im vorigen Abschnitt vorgestellte Mini-Programm dient als Skelett für alle folgenden Windows-Programme. Es wird (von Kommentaren befreit) noch einmal angegeben:

```
/*  Windows-Skelett-Programm (winskel.c)
    ====================================  */

#include <windows.h>

LONG FAR PASCAL WndProc (HWND , UINT , UINT , LONG) ;

int PASCAL WinMain (HANDLE hInstance    , HANDLE hPrevInstance ,
                    LPSTR  lpszCmdParam , int    nCmdShow)
{
  MSG      msg      ;
  HWND     hwnd     ;
  WNDCLASS wndclass ;

  if (!hPrevInstance)
    {
      wndclass.style         = CS_HREDRAW | CS_VREDRAW  ;
      wndclass.lpfnWndProc   = WndProc ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
      wndclass.hInstance     = hInstance   ;
      wndclass.hIcon         = LoadIcon (NULL , IDI_APPLICATION) ;
      wndclass.hCursor       = LoadCursor (NULL , IDC_ARROW)  ;
      wndclass.hbrBackground = GetStockObject (LTGRAY_BRUSH)  ;
      wndclass.lpszMenuName  = NULL           ;
      wndclass.lpszClassName = "WndClassName" ;

      RegisterClass (&wndclass) ;
    }

  hwnd = CreateWindow ("WndClassName" , "Ueberschrift" ,
                 WS_OVERLAPPEDWINDOW , CW_USEDEFAULT  ,
                       CW_USEDEFAULT , CW_USEDEFAULT  , CW_USEDEFAULT ,
                       NULL  ,  NULL , hInstance      , NULL) ;

  ShowWindow   (hwnd , nCmdShow) ;
  UpdateWindow (hwnd)            ;

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

  return msg.wParam ;
}

LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)
{
  switch (message)
  {
    case WM_DESTROY:
      PostQuitMessage (0) ;
      return 0            ;
  }
  return DefWindowProc (hwnd , message , wParam , lParam) ;
}
```

- In WinMain wurde der Aufruf von **UpdateWindow** ergänzt (**ShowWindow** füllt das Fenster mit Hintergrundfarbe, **UpdateWindow** sorgt für das Zeichnen des Inhalts).
- In der Schleife der Funktion WinMain wurde der Aufruf der Funktion **TranslateMessage** ergänzt. Diese übersetzt spezielle von der Tastatur kommende Botschaften, läßt die übrigen Botschaften ungeändert, wird in den meisten Programmen eigentlich nicht benötigt, schadet aber in keinem Fall und sollte vorsichtshalber eingebunden werden.
- Die Namen für die Variablen, die vom Benutzer frei gewählt werden dürfen, wurden an die Konventionen angepaßt, an die sich erstaunlich viele Windows-Programmierer, die Programme veröffentlicht haben, halten. Es sind eigentlich nur zwei immer wieder verwendeten Strategien zur Konstruktion von Namen: Wenn nur eine Variable eines speziellen Typs benötigt wird, verwendet man den Typnamen selbst, allerdings mit Kleinbuchstaben, in **windows.h** sind die mit **typedef** erzeugten Typnamen in der Regel durch Großbuchstaben gekennzeichnet (Beispiel: **WNDCLASS wndclass**). Ansonsten wird die sogenannte "ungarische Notation" (nach dem Herkunftsland des Erfinders dieser Variante) favorisiert: Einem "sprechenden" Variablennamen, der mit einem Großbuchstaben beginnt, werden in Kleinbuchstaben Typ-Informationen vorangestellt, z. B. **lParam** für einen Parameter vom Typ **LONG** (zugegebenermaßen ist allerdings **lpszCmdParam** für einen "Parameter aus der Kommandozeile" vom Typ "Long Pointer auf einen Zero-Terminated-String" etwas gewöhnungsbedürftig).
- Nachfolgend wird die Definitions-Datei **winskel.def** angegeben, die sich nur durch den unter Schlüsselwort **NAME** angegebenen Namen und den Namen der Fenster-Funktion (unter dem Schlüsselwort **EXPORTS**) von der im vorigen Abschnitt angegebenen Datei **miniwin.def** unterscheidet:

```
; Der NAME (hier: WINSKEL) und die Angaben nach dem Schluesselwort
; EXPORTS muessen dem aktuellen Problem angepasst werden, alle uebrigen
; Zeilen koennen fuer die meisten Programme ungeaendert uebernommen
;  werden

NAME      WINSKEL                  ; Das Schluesselwort "NAME" legt fest,
                                   ; dass ein ausfuehrbares Programm (und
                                   ; keine LIBRARY) erzeugt werden soll

; Die folgenden Zeilen entsprechen den Default-Vorgaben von MS-Visual-C++
; 1.5 und sind sinnvoll fuer die meisten "normalen" Programme:

EXETYPE   WINDOWS
CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD MOVEABLE MULTIPLE
HEAPSIZE  1024

; Nach dem Schluesselwort EXPORTS muessen alle Funktionen aufgelistet
; werden, die von Windows direkt angesprochen werden (Ausnahme: WinMain
; braucht nicht aufgefuehrt zu werden), weil diese Funktionen mit selbst
; zu waehlenden Namen versehen werden koennen und im eigentlichen
; Programm nicht aufgerufen werden, so dass der Linker gar nicht wissen
; kann, dass sie gebraucht werden.
;
; In diesem Fall ist es nur die "Fensterfunktion".

EXPORTS
 WndProc @1
```

9.5 Text- und Graphik-Ausgabe, der "Device Context"

Windows hat eine graphische Benutzer-Oberfläche, und deshalb ist Text auch immer Graphik, die C-Funktion **printf** hat also ausgedient. Windows definiert eigene Funktionen für die Ausgabe von Text und Graphik, die im sogenannten **GDI** ("Graphics Device Interface") zusammengefaßt sind.

Das **Graphics Device Interface** (GDI) gestattet dem Programmierer eine weitgehend geräteunabhängige Codierung der Text- und Graphikausgabe. Zwischen Anwenwenderprogramm und Gerätetreiber wird von Windows ein **Device Context** gelegt. Dies ist eine (ziemlich umfangreiche) interne Windows-Datenstruktur, die alle denkbaren Zeichenattribute (Farben, "Zeichenstift"-Positionen, Füllmuster, "Clipping"-Gebiete, ...) enthält, die für die Ausgabeaktion benutzt werden.

Die meisten Attribute in einem Device Context sind sinnvoll mit Standardwerten vorbelegt, so daß der Programmierer vor einer Ausgabe

- einen Device Context von Windows anfordern ...
- und nur die für die Aktion relevanten Attribute speziell definieren bzw. anpassen muß. Dafür stehen ihm im GDI zahlreiche Funktionen zur Verfügung.

9.5.1 Die Botschaft WM_PAINT, Programm "Hello, Winworld"

Die Botschaft **WM_PAINT** nimmt unter den zahlreichen Botschaften, die von Windows an ein Programm gegeben werden, eine gewisse Sonderstellung ein (im Gegensatz zur Priorität, die ihr zugeordnet ist, sie wird schlicht in die Warteschlange für das Programm eingereiht und nicht etwa direkt an die Fensterfunktion gesendet). Sie wird immer dann ausgelöst, wenn der Inhalt eines Fensters (oder ein Teil davon) "ungültig" geworden ist, z. B.

- durch die Freigabe eines Bereichs, der vorübergehend von einem anderen Fenster überdeckt war,
- durch das Verändern der Fenstergröße (beim Verkleinern braucht der verbleibende Rest nicht zwingend "ungültig" zu sein, wenn allerdings - wie in **winskel.c** - dem Fenster der Stil **wndclass.style = CS_HREDRAW | CS_VREDRAW** zugeordnet wird, kommt die Botschaft WM_PAINT bei jeder Größenänderung),
- natürlich beim Anlegen eines Fensters (deshalb wird in **winskel.c** die Funktion **UpdateWindow** aufgerufen, die nicht viel mehr tut, als eine Botschaft WM_PAINT auszulösen),
- durch das "Scrollen" des Fensterinhalts mit Hilfe der (später zu behandelnden) "Bildlaufleisten",

- durch das gezielte "Ungültigmachen" eines Fensterbereichs durch den Programmierer mit der (ebenfalls später zu besprechenden) Funktion **InvalidateRect**.

Bei der Bearbeitung der Botschaft WM_PAINT wird die Fensterfunktion in der Regel den gesamten aktuellen Fensterinhalt neu zeichnen. Dafür wird ein Device Context benötigt, der in diesem Fall neben dem Bezug zum Bildschirm-Treiber auch noch einen Bezug zum speziellen Fenster haben muß.

Nur für das Bearbeiten der Botschaft WM_PAINT steht eine spezielle Variante zur Verfügung, einen **Device Context** anzufordern und wieder freizugeben:

Alle Zeichenaktionen im Zusammenhang mit der Botschaft WM_PAINT werden von

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

"eingerahmt".

- **BeginPaint** wird mit dem Handle **hwnd** auf das zu bearbeitende Fenster und dem Pointer auf eine Struktur **ps** vom Typ **PAINTSTRUCT** aufgerufen und liefert einen "Handle of Device Context" **hdc** ab (und löscht den Fensterinhalt durch Überzeichen mit dem definierten Fenster-Hintergrund).
- Der Handle **hdc** ist für alle nachfolgenden Zeichenroutinen der Bezug für das Ziel der Zeichenaktionen.
- **EndPaint** gibt den Device Context wieder frei und gibt dem Fenster den Status "aktualisiert".

Die Komponenten der Struktur vom Typ **PAINTSTRUCT** sind für den Programmierer in der Regel von geringem Interesse, sie werden von den GDI-Funktionen benötigt. Wichtig ist, daß in der Fensterfunktion eine Struktur dieses Typs vereinbart ist (ebenso wie ein "Handle of Device Context" vom Typ **HDC**).

Das nachfolgende Programm ist die Windows-Variante des "Hello-World"-Klassikers und demonstriert die Bearbeitung der Botschaft WM_PAINT. Da in WinMain (neben dem Kommentar am Anfang) nur ein Parameter eines Funktionsaufrufs gegenüber **winskel.c** geändert wurde, wird WinMain nur auszugsweise gelistet:

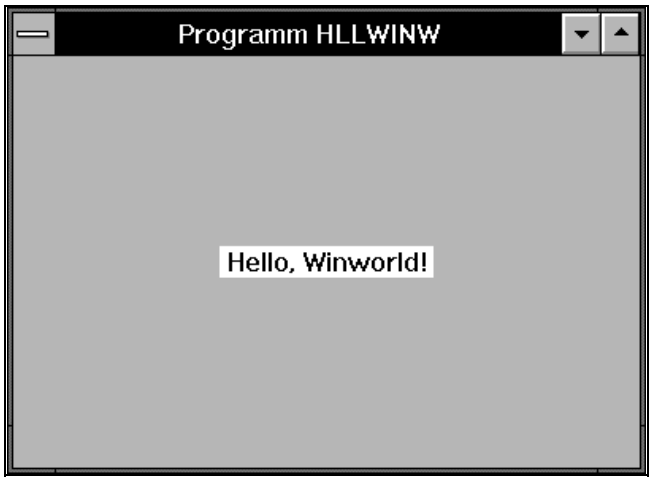
```
/*  Ausgabe von Text ueber "Device Context" (hllwinw.c)
    ================================================  */

/*  Das Programm verwendet das Skelettprogramm winskel.c, das nur an
    zwei Stellen geaendert bzw. erweitert wird:

    *  In WinMain wird die Funktion CreateWindow mit einer dem
       Programmnamen angepassten Fenster-Ueberschrift aufgerufen.
```

```
    *  In WndProc wird die Botschaft WM_PAINT zusaetzlich
       ausgewertet.

    Demonstriert werden mit diesem Programm

    *  das Arbeiten mit einem "Device Context",
    *  das Zusammenspiel der beiden Windows-Funktionen BeginPaint
       und EndPaint,
    *  die Funktion GetClientRect ("Netto"-Abmessungen des Fensters),
    *  die Funktion DrawText.                                          */

...

int PASCAL WinMain (...

...

  hwnd = CreateWindow ("WndClassName" , "Programm HLLWINW" ,
                  WS_OVERLAPPEDWINDOW , CW_USEDEFAULT  ,
                        CW_USEDEFAULT , CW_USEDEFAULT  , CW_USEDEFAULT ,
                          NULL , NULL , hInstance      , NULL) ;

...


LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)
{
  HDC          hdc  ;  /* ... ist ein "Handle of Device Context"          */
  PAINTSTRUCT  ps   ;  /* ... ist eine Struktur mit allen Informationen,
                          die Windows fuer das Zeichnen in einem
                          Fenster auswertet                               */
  RECT         rect ;  /* ... fuer die Aufnahme der Abmessungen der
                          Zeichenflaeche                                  */

  switch (message)
  {
    case WM_PAINT:     /* ... wird u. a. von UpdateWindow oder beim Aendern
                          der Fenstergroesse erzeugt, Fensterinhalt wird
                          neu gezeichnet, eine Zeichenaktion sollte immer
                          von BeginPaint und EndPaint umschlossen sein: */


      hdc = BeginPaint (hwnd , &ps) ;
                       /* ... liefert "Handle of Device Context" und die
                          zugehoerigen PaintStruktur-Informationen,
                          loescht ausserdem den Fensterinhalt durch
                          Ueberzeichnen mit dem durch den hbrBackground-
                          Parameter der Fensterklasse festgelegten
                          Muster (in winskel.c: LTGRAY_BRUSH)             */

      GetClientRect (hwnd , &rect) ;
                       /* ... liefert die "Netto"-Abmessungen des Fensters  */

      DrawText (hdc , " Hello, Winworld! " , -1 , &rect ,
                                DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
                       /* ... schreibt einen durch die ASCII-Null
                          abgeschlossenen Text (gekennzeichnet durch
                          die -1) in ein Rechteck des durch Device Context
                          definierten Geraets, der einzeilige Text wird
                          im Rechteck horizontal und vertikal zentriert */

      EndPaint (hwnd , &ps) ;
                       /* ... raeumt auf und gibt Device Context frei       */


      return 0           ;
```

```
    case WM_DESTROY:

      PostQuitMessage (0) ;
      return 0            ;
  }

  return DefWindowProc (hwnd , message , wParam , lParam) ;
}

/*  Die Funktion GetClientRect liefert die "Netto"-Abmessungen des Fensters
    (ohne Rand, Kopfleiste usw.), das durch den Handle hwnd gekennzeichnet
    ist, in einer Struktur vom Typ RECT ab. Diese wird in windows.h wie
    folgt definiert:

                          typedef struct tagRECT
                              {
                                  int left   ;
                                  int top    ;
                                  int right  ;
                                  int bottom ;

                              } RECT ;

    Die int-Werte sind "Pixel" und beziehen sich auf das Standard-Koordinaten-
    system, das seinen Ursprung in der linken oberen Ecke des Fensters hat, so
    dass von GetClientRect fuer left und top immer die Werte 0 und fuer right
    und bottom die Breite bzw. die Hoehe des Zeichenbereichs geliefert werden.

    In diesem Programm wird die Rechteck-Struktur nicht direkt ausgewertet,
    sondern an DrawText weitergereicht.                                    */

/*  Die Funktion DrawText ist eine von mehreren Textausgabe-Funktionen des
    GDI. Typisch fuer alle GDI-Funktionen ist der Bezug auf den Device Context
    (ueber Handle hdc), der das "Wohin und Wie" festlegt. Der auszugebende
    Text (2. Argument) wird als "Zero-Terminated-String" (abgeschlossen
    durch ASCII-Null wie alle String-Konstanten in C) durch die -1 auf der
    3. Position ausgewiesen. Das vierte Argument legt den Rechteckbereich
    fest, in den geschrieben wird, schliesslich sind die drei durch das
    "Logische Oder" verknuepften Parameter in

      DrawText (hdc , " Hello, Winworld! " , -1 , &rect ,
                                DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

    dafuer zustaendig, die Art der Ausgabe in den Rechteckbereich
    festzulegen (hier: "Eine Zeile, zentriert in beiden Richtungen", die
    "DT_...-Konstanten" sind ebenfalls in windows.h definiert).            */
```

Die nebenstehende Abbildung zeigt die Ausgabe des Programms. Der Text wird bei jeder Änderung der Fenstergröße neu geschrieben und ist dadurch immer zentriert bezüglich der aktuellen Fenstergröße.

Da im angeforderten Device Context kein Parameter geändert wurde, hat **DrawText** die Standardwerte verwendet, z. B. "Schwarze Schrift auf weißem Hintergrund" mit dem Standard-Font und den voreingestellten Werten für die Größe der Textzeichen.



Ausgabe des Programms **hllwinw.c**

9.5.2 Zeichnen mit MoveTo und LineTo, Programm "rosette1.c"

Das nachfolgende Programm zeigt, daß Graphik-Ausgabe unter Windows nach der gleichen Strategie wie die im vorigen Abschnitt demonstrierte Text-Ausgabe erfolgt:

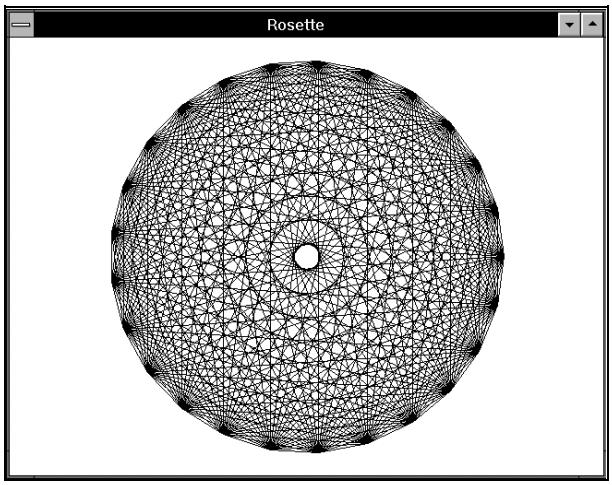
```
/*  Graphik-Ausgabe ueber "Device Context" (rosette1.c)
    ===================================================   */

/*  Auf einem (nicht gezeichneten) Kreis, dessen Durchmesser 9/10 der
    kleineren Fensterabmessung ist, werden 25 Punkte gleichmaessig verteilt
    und jeder Punkt mit jedem anderen durch eine Gerade verbunden (es
    werden 600 gerade Linien gezeichnet).

    Das Programm demonstriert die typischen Zeichenaktionen, wie sie beim
    Eintreffen der Nachricht WM_PAINT ausgefuehrt werden sollten:

    *  Aktionen werden von BeginPaint und EndPaint eingerahmt.

    *  Fuer den von BeginPaint gelieferten "Device context" werden der
       Ursprung des Koordinatensystems mit SetViewportOrg festgelegt und ...

    *  mit MoveTo und LineTo die "Klassiker" der Computergraphik verwendet.

    *  Zusaetzlich wird die Botschaft WM_SIZE bearbeitet, wobei die beiden
       in windows.h definierten Makros LOWORD und HIWORD verwendet werden.  */

#include <windows.h>
#include <math.h>             /* ... weil Winkelfunktionen verwendet werden */

LONG FAR PASCAL WndProc (HWND , UINT , UINT , LONG) ;

int PASCAL WinMain (HANDLE hInstance    , HANDLE hPrevInstance ,
                    LPSTR  lpszCmdParam , int    nCmdShow)
{
  ...

     wndclass.hbrBackground = GetStockObject (WHITE_BRUSH)  ;
                         /* ... fuer einen weissen Hintergrund des Fensters */
  ...


  hwnd = CreateWindow ("WndClassName" , "Rosette" ,

  ...


LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)
{
  HDC         hdc ;
  PAINTSTRUCT ps  ;
  static int  cxClient , cyClient , nPoints = 25 ; /* .. Kommentar am Ende */
  double      Radius , dPhi ;    /* .. Radius des Kreises, Winkel-Schritte */
  int         i , j , xs , ys ;

     switch (message)
        {
          case WM_SIZE :                    /* ... siehe Kommentar am Ende */

               cxClient = LOWORD (lParam) ;  /* .. sind die Abmessungen des */
               cyClient = HIWORD (lParam) ;  /*    Zeichen-Fensters (Pixel) */

               return 0 ;

          case WM_PAINT :
```

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

                SetViewportOrg (hdc, cxClient / 2 , cyClient / 2) ;
                  /* .. legt Ursprung des Koordinatensystems in Fenstermitte */

                Radius = ((cxClient > cyClient) ? cyClient : cxClient) *.45 ;
                dPhi   = atan (1.) * 8. / nPoints ;  /*   pi * 2 / nPoints   */

                for (i = 0 ; i < nPoints ; i++)
                  {
                     xs = (int) (Radius * cos (dPhi * i) + .5) ;
                     ys = (int) (Radius * sin (dPhi * i) + .5) ;

                     for (j = 0 ; j < nPoints ; j++)  /* Achtung! Die GDI-   */
                       {                              /* Funktionen verwen-  */
                          if (j != i)                 /* den ausschliesslich */
                            {                         /* Integer-Koordinaten */
                               MoveTo (hdc , xs , ys) ;
                               LineTo (hdc ,
                                       (int) (Radius * cos (dPhi * j) + .5)  ,
                                       (int) (Radius * sin (dPhi * j) + .5)) ;
                                  /* ... werden am Programm-Ende beschrieben */

                            }
                       }
                  }

                EndPaint (hwnd, &ps) ;

                return 0 ;

           case WM_DESTROY :

                PostQuitMessage (0) ;

                return 0 ;
          }
       return DefWindowProc (hwnd, message, wParam, lParam) ;
}


/*  Die Botschaft WM_SIZE wird erzeugt, wenn sich die Groesse eines Fensters
    aendert (natuerlich auch beim Erzeugen des Fensters). Mitgeliefert wird
    im Parameter lParam die Fenstergroesse, der als LONG-Parameter beide
    int-Werte (Breite und Hoehe) enthaelt. Zum "Zerlegen" eines LONG-
    Parameters "in zwei Teile" werden in windows.h zwei Makros definiert:

    LOWORD (lParam) liefert die (in den beiden niedrigwertigen Bytes
    gespeicherte) Breite, HIWORD (lParam) liefert die (in den beiden
    hoeherwertigen Bytes gespeicherte) Hoehe des Zeichenbereichs (Pixel).

    Da die so ermittelten (und in den Variablen cxClient bzw. cyClient
    abgelegten) Fensterabmessungen erst bei einem nachfolgenden
    Aufruf von WndProc (Bearbeitung der Botschaft WM_PAINT) verwendet
    werden, muessen die Werte beim Verlassen der Funktion WndProc erhalten
    bleiben. Dafuer bieten sich in C folgende Moeglichkeiten an:

    *  Die Variablen cxClient und cyClient werden global (ausserhalb aller
       Funktionen) vereinbart (wenn vermeidbar, sollte man allerdings auf
       globale Variablen verzichten).

    *  Die Variablen cxClient und cyClient werden der Speicherklasse
       static zugeordnet. Dann verlieren sie auch beim Verlassen einer
       Funktion nicht ihren Wert (wie die Variablen der Speicherklasse
       auto, dies ist die Standard-Speicherklasse fuer Variablen in C)
       und stehen bei einer nachfolgenden Abarbeitung der Funktion mit
       ihren alten Werten zur Verfuegung.                                  */
```

```
/*  Die Funktion

                SetViewportOrg (hdc , xViewOrg , yVieworg) ;

    verschiebt den Ursprung des Koordinatensystems aus der linken oberen
    Ecke um die als int-Werte (Pixel) anzugebenden Argumentwerte von
    xViewOrg und yViewOrg (nach rechts bzw. unten). Die Orientierung der
    Achsen (nach rechts bzw. unten) bleibt dabei erhalten. Alle nachfolgenden
    Ausgaben ueber den durch hdc definierten Device Context beziehen sich
    auf das verschobene Koordinatensystem.                                  */

/*  Die Anweisung

          Radius = ((cxClient > cyClient) ? cyClient : cxClient) *.45 ;

    errechnet einen Radius, der die 0.45-fache Laenge der jeweils kleineren
    Zeichenflaechen-Abmessung hat (die Zeichnung wird so passend zu den
    Fensterabmessungen skaliert).

    In windows.h sind uebrigens zwei Makros min(a,b) und max(a,b) definiert,
    so dass die Anweisung auch einfacher als

                Radius = min (cxClient , cyClient) *.45 ;

    geschrieben werden koennte.

    Die Positionen, die (schon wegen der Verwendung der Winkelfunktionen)
    saemtlich zunaechst mit double-Variablen errechnet werden, muessen
    auf int-Wert "gecastet" werden, weil alle GDI-Funktionen int-Werte
    erwarten (eine erhebliche Schwaeche).                                   */


/*  Die beiden Funktionen

                      MoveTo (hdc , xStart , yStart) ;
                      LineTo (hdc , xEnd   , yEnd)   ;

    arbeiten wie alle Funktionen mit diesen Namen in Graphik-Systemen:
    MoveTo postiert den imaginaeren Zeichenstift an einem (mit int-Werten
    anzugebenden) Punkt, ohne eine Zeichenaktion dabei auszufuehren. LineTo
    zeichnet eine gerade Linie von der aktuellen Position (durch einen
    vorangegangenen Aufruf von MoveTo oder LineTo hinterlassen) zu der
    (mit int-Werten anzugebenden) End-Position.                             */
```

Die nebenstehende Abbildung zeigt die Ausgabe des Programms.

Bei jeder Änderung der Fenstergröße werden die Fensterabmessungen aktualisiert (Botschaft WM_SIZE), die beim anschließenden Neuzeichnen des Fensterinhalts genutzt werden, um den Radius der Rosette anzupassen, so daß das Bild (auch bei einem Fenster, dessen Höhe größer als die Breite ist) immer in das Fenster "paßt".



Ausgabe des Programms **rosette1.c**

9.6 Maus-Botschaften, Programm "mouse1.c

Die Maus ist das bevorzugte Eingabegerät für Windows-Programme. Das nachfolgende Beispiel-Programm demonstriert die Auswertung einiger Botschaften, die von der Maus an ein Fenster geschickt werden.

Von WinMain werden nur die Zeilen angegeben, die sich vom Skelettprogramm **winskel.c** unterscheiden:

```
/*  Auswerten von Maus-Eingaben (mouse1.c)
    ======================================  */

/*  In das Hauptfenster des Programms werden bei gedrueckter linker
    Maustaste alle Mausbewegungen als Linien eingezeichnet. Beim
    Druecken der rechten Maustaste wird das Fenster geloescht.

    Demonstriert werden mit diesem Programm

    *  die Auswertung der Botschaften WM_MOUSEMOVE und WM_RBUTTONDOWN,

    *  die Funktion InvalidateRect.                                    */


int PASCAL WinMain (...

...

      wndclass.hbrBackground = GetStockObject (WHITE_BRUSH)  ;

...

  hwnd = CreateWindow ("WndClassName" , "Zeichnen mit der Maus" ,

...


LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)

{
  HDC         hdc ;
  PAINTSTRUCT ps  ;
  static int  xs = 0 , ys = 0 , xe = 0 , ye = 0 , draw = 0 ;
                      /* ... muessen "static" vereinbart werden, weil die
                         eigentliche Zeichenaktion erst beim zweiten
                         Aufruf von WndProc realisiert wird               */

  switch (message)
  {
    case WM_MOUSEMOVE:
                      /* ... ist die Nachricht, dass sich die Maus bewegt
                         hat. Mit dieser Nachricht werden folgende
                         Informationen uebergeben:

                         wParam enthaelt bitweise verschluesselt den Zustand
                         der drei Maustasten und der Shift- und der
                         Ctrl-Taste, die (siehe folgendes Beispiel) mit den
                         Masken MK_LBUTTON, MK_MBUTTON, MK_RBUTTON, MK_SHIFT
                         und MK_CONTROL herausgefiltert werden koennen.

                         lParam enthaelt die Cursor-Position (in Pixel-
                         Koordinaten): Horizontale Koordinate ist das
                         niederwertige Wort, vertikale Koordinate ist das
                         hoeherwertige Wort.                              */
```

```
      xs = xe ;
      ys = ye ;                    /* ... sichert alte Cursor-Position      */

      xe = LOWORD (lParam) ;       /* LOWORD und HIWORD sind Makros aus     */
      ye = HIWORD (lParam) ;       /* windows.h, vgl. Programm rosette1.c   */
                     /* Die in lParam abgelieferten Koordinaten (Pixel)
                        beziehen sich auf das in der linken oberen Bild-
                        schirmecke liegende Koordinatensystem.             */

      if (wParam & MK_LBUTTON)     /* ... ist linke Maustaste gedrueckt, .. */
       {
        InvalidateRect (hwnd , NULL , FALSE) ; /* ... und es wird die
                        Nachricht WM_PAINT abgesetzt, der dritte Parameter
                        in InvalidateRect ist das "Erase-Flag", das hier
                        mit FALSE anzeigt, dass vor der Zeichenaktion das
                        Fenster NICHT geloescht werden soll                */
        draw = 1 ;
       }

      return 0 ;

    case WM_RBUTTONDOWN:            /* ... wurde rechte Maustaste gedrueckt */
       {
        InvalidateRect (hwnd , NULL , TRUE) ;    /* ... loescht das Fenster */
       }

      return 0 ;

    case WM_PAINT:    /* Es wird mit folgender Strategie gearbeitet: Wenn
                         der Parameter draw den Wert 1 hat, werden die
                         beiden letzten registrierten Maus-Positionen durch
                         eine Gerade verbunden. Wenn draw den Wert 0 hat,
                         wird nur das Funktionenpaar BeginPaint/EndPaint
                         ausgefuehrt (Fenster loeschen und als "aktualisiert"
                         kennzeichnen).                                    */


      hdc = BeginPaint (hwnd , &ps) ;

      if (draw)
        {
          MoveTo (hdc , xs , ys) ;   /* ... zeichnet eine Gerade von alter  */
          LineTo (hdc , xe , ye) ;   /*     nach neuer Cursor-Position      */
          draw = 0 ;
        }

      EndPaint (hwnd , &ps) ;

      return 0 ;

    case WM_DESTROY:

      PostQuitMessage (0) ;
      return 0            ;
  }

  return DefWindowProc (hwnd , message , wParam , lParam) ;
}


/*  Die wichtigsten Botschaften, die fuer ein Fenster bestimmte
    Mausereignisse signalisieren, sind:

       WM_LBUTTONDOWN     -->   Linker    Mausknopf gedrueckt,
       WM_RBUTTONDOWN     -->   Rechter   Mausknopf gedrueckt,
       WM_MBUTTONDOWN     -->   Mittlerer Mausknopf gedrueckt,
       WM_LBUTTONUP       -->   Linker    Mausknopf geloest,
       WM_RBUTTONUP       -->   Rechter   Mausknopf geloest,
```

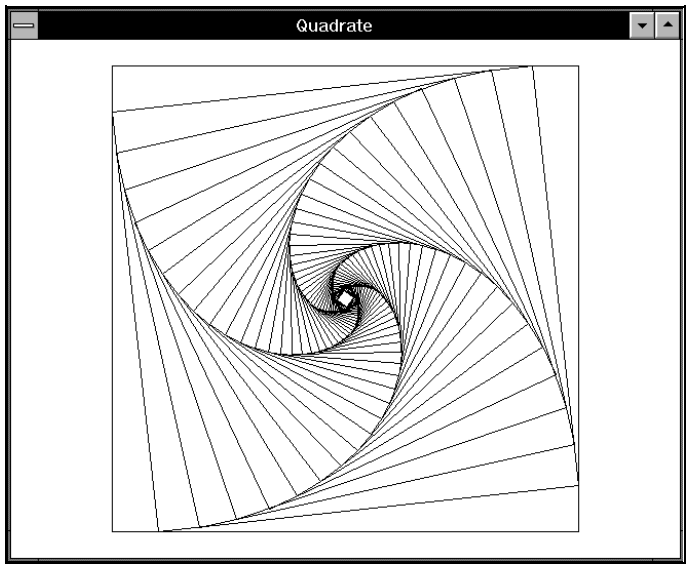
```
   WM_MBUTTONUP       -->   Mittlerer Mausknopf geloest,
   WM_LBUTTONDBCLCK   -->   Linker    Mausknopf doppelt gedrueckt,
   WM_RBUTTONDBCLCK   -->   Rechter   Mausknopf doppelt gedrueckt,
   WM_MBUTTONDBCLCK   -->   Mittlerer Mausknopf doppelt gedrueckt,

   WM_MOUSEMOVE       -->   Maus wurde im Fensterbereich bewegt.

   Bei der Botschaft WM_MOUSEMOVE koennen der Zustand des rechten bzw.
   linken Mausknopfes und der Zustand der Shift- bzw. Ctrl-Taste der
   Tastatur mit den im Programmkommentar genannten Masken aus wParam heraus-
   gefiltert werden, die aktuellen Koordinaten sind in lParam zu finden.

   "Doppelklicks" kann ein Fenster nur empfangen, wenn es bei der
   Definition der Fensterklasse (mit RegisterClass) im Fensterstil angegeben
   wurde, z. B.:

            wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS ;        */

/*  Die Funktion InvalidateRect wird mit 3 Argumenten aufgerufen, z. B.:

               InvalidateRect (hwnd , NULL , TRUE) ;

   ... loescht das durch den Handle hwnd gekennzeichnete Fenster und schickt
   ihm die Botschaft WM_PAINT (Inhalt erneuern). Wenn als drittes Argument
   FALSE angegeben ist, wird (ohne Loeschen des Fensterinhalts) nur WM_PAINT
   abgesetzt.

   Das zweite Argument zeigt mit dem Wert NULL an, dass der gesamte Fenster-
   bereich "ungueltig" ist, im allgemeinen Fall kann dort eine Struktur vom
   Typ RECT eingesetzt werden (vgl. Kommentar des Programms hllwinw.c im
   Abschnitt 9.5.1), die nur einen Teil des Rechteckbereichs als "ungueltig"
   kennzeichnet.

   Die Konstanten TRUE, FALSE und NULL findet man in windows.h:

                        typedef int BOOL ;
                        #define     FALSE  0
                        #define     TRUE   1
                        #define     NULL   0

   Die meisten Programme werden bei einer Botschaft WM_PAINT ohnehin den
   gesamten Fensterinhalt erneuern, weil der Programmieraufwand fuer das
   Neuzeichnen jeweils eines Bereichs recht gross sein kann. Trotzdem ist
   eine Rechteckangabe beim Aufruf von InvalidateRect sinnvoll, denn
   Windows selbst sorgt dafuer, dass beim Neuzeichnen eines Fensterinhalts
   tatsaechlich nur der "ungueltige" Bereich neu gezeichnet wird, was
   erhebliche Zeitersparnis bedeuten kann.                                 */
```

Die nebenstehende Skizze zeigt den Versuch, in das Fenster mit der Maus eine Zeichnung einzubringen.

Man beachte, daß das Programm **mouse1.c** den Fensterinhalt nicht verwaltet und deshalb auch nicht erneuern kann. Wenn also von Windows eine Botschaft WM_PAINT geschickt wird, weil sich z. B. die Fenstergröße geändert hat, wird im Programm nur das Funktionenpaar **BeginPaint/EndPaint** ausgeführt, was zum Löschen des Fensters führt.

Aufgabe 9.1: Es ist ein Programm **quadrat1.c** zu schreiben, das in das Hauptfenster ein Quadrat zeichnet, darin ein weiteres Quadrat, dessen Eckpunkte auf den vier Seiten des ersten Quadrates liegen, jeweils **1/10** der Seitenlänge von den Eckpunkten des ersten Quadrates entfernt. Darin soll ein weiteres Quadrat nach der gleichen Vorschrift gezeichnet werden usw., insgesamt 50 Quadrate.

Die Größe der Zeichnung soll sich einer Änderung der Fenstergröße anpassen.



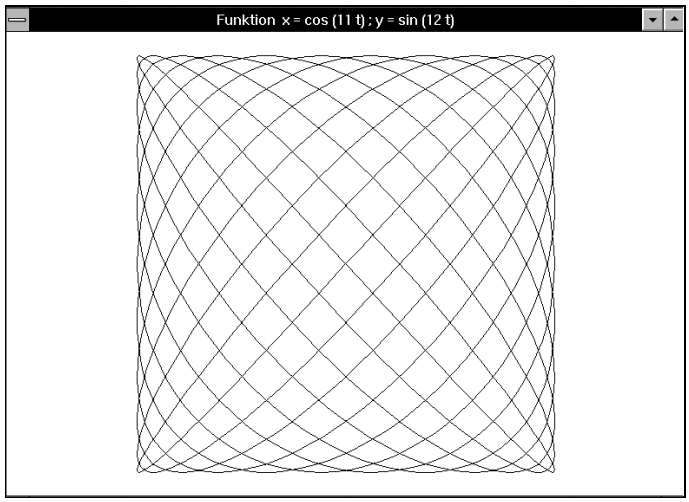
Aufgabe 9.2: Es ist ein Programm **paramet1.c** zu schreiben, das die in Parameterdarstellung gegebene Funktion

$$x = a \cos(11\,t)$$
$$y = a \sin(12\,t)$$

im Bereich

$$0 \le t \le 2\pi$$

darstellt (a ist stets so zu wählen, daß die kleinere Fensterabmessung zu 90% ausgefüllt wird).



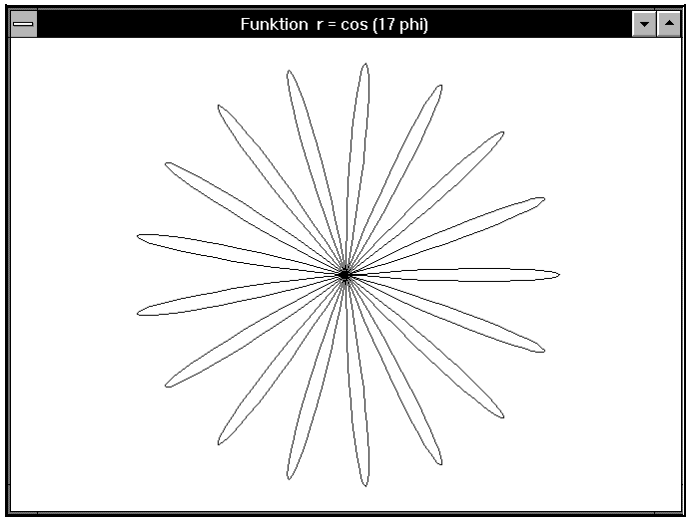
Aufgabe 9.3: Es ist ein Programm **polar1.c** zu schreiben, das die in Polarkoordinaten gegebene Funktion

$$r = a \cos(17\,\varphi)$$

im Bereich

$$0 \le \varphi \le 2\pi$$

darstellt (a ist stets so zu wählen, daß die kleinere Fensterabmessung zu 90% ausgefüllt wird).



Eigentlich ist es ganz einfach: Wenn der Benutzer (über Windows) dem Programm signalisiert, daß es beendet werden soll, dann schickt das Programm an Windows die Bitte, ihm die Botschaft zu schicken, daß es sich beenden soll.

10 Ressourcen

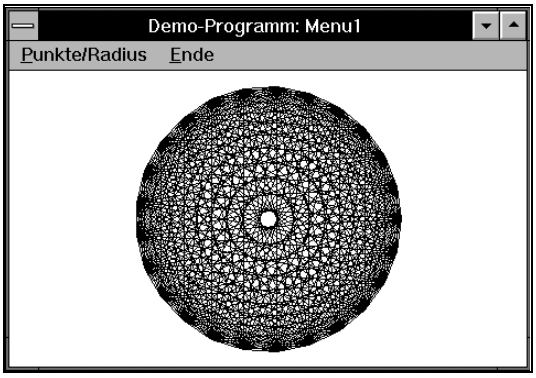
Als **Ressourcen** werden Graphiken und Texte bezeichnet, die das äußere Erscheinungsbild eines Windows-Programms weitgehend bestimmen können, aber in gesonderten Dateien (separat vom eigentlichen Quelltext des Programms) definiert werden. Auf diese Weise kann die Benutzer-Schnittstelle auf recht einfache Weise verändert werden (beim Übertragen des Programms in eine andere Sprache braucht z. B. bei konsequentem Arbeiten mit Ressourcen der Programm-Code nicht geändert zu werden).

Der Programmierer kann **Zeichenketten, Menüs, Dialoge, Datensammlungen beliebiger Art, Abkürzungsbefehle, Zeichensätze, Icons, Bitmaps und Cursorformen** in ASCII-Dateien (übliche Extension: **.rc**) definieren und sich im Programm auf diese Ressourcen beziehen. Die Dateien werden von einem "Ressource-Compiler" übersetzt (zu MS-Visual-C gehört z. B. **rc.exe**), der im allgemeinen auch das Anbinden der übersetzten Datei an die EXE-Datei übernimmt (diese Aktion erfolgt **nach** dem Linken des Objectcodes, nachdem also das EXE-Programm bereits entstanden ist und wird nicht vom Linker erledigt).

In diesem Kapitel wird das Definieren von Menüs, Zeichenketten, Dialogboxen, Icons und Cursorformen demonstriert, und es wird gezeigt, wie vom Programm auf die definierten Ressourcen Bezug genommen wird. Dabei wird zunächst alles "von Hand" erledigt, um die Zusammenhänge zu verdeutlichen. In den Abschnitten 10.3.3 und 10.4.1 wird dann angedeutet, daß zu einem Windows-Entwicklungssystem im allgemeinen sehr leistungsfähige Tools für das Erzeugen von Ressourcen gehören.

10.1 Menü und Message-Box, Programm "menu1.c"

Das Programm **menu1.c** zeigt an einem besonders einfachen Beispiel (Definition eines aus nur zwei Angeboten bestehenden Menüs) das Zusammenspiel einer Ressource, die in der Datei **menu1.rc** beschrieben wird, mit dem Programm. Es ist eine nur geringfügige Erweiterung des Programms rosette1.c aus dem Abschnitt 9.5.2 (Ziel ist das zusätzliche Erzeugen der Menüleiste, vgl. nebenstehende Abbildung), so daß vom Programmcode nur wenige Zeilen aufgelistet



werden. Die Definition des Menüs befindet sich in der nachfolgend angegebenen Ressource-Datei, die nur durch den ausführlichen Kommentar (wird jeweils durch das Semikolon eingeleitet) einen Umfang hat, der dem kleinen Problem nicht angemessen ist:

```
; Definition eines ganz einfachen Menues:

Menu1_Menu MENU   ; Der Name "Menu1_Menu" vor dem Schluesselwort MENU ist
                  ; der Bezug fuer das Fenster, zu dem das Menu gehoeren
                  ; soll, und wird beim Definieren der Window Class mit
                  ;
                  ;           wndclass.lpszMenuName = "Menu1_Menu"
                  ;
                  ; der nachfolgenden Definition zugeordnet:
{
       MENUITEM "&Punkte/Radius"     ,  10
       MENUITEM "&Ende"              ,  20  ; ... definieren zwei
                  ; einfache Menue-Items fuer die Menueleiste am oberen
                  ; Fensterrand. Mit dem '&' vor einem Zeichen wird
                  ; dieses als zu unterstreichendes Zeichen festgelegt
                  ; (damit kann dieses Menueangebot mit der
                  ; entsprechenden Taste gewaehlt werden). Die Zahl nach
                  ; dem Komma legt den Wert des Parameters wParam fest,
                  ; der mit der Botschaft WM_COMMAND uebergeben wird.
}
```

Resource-Datei menu1.rc

Da das erste Menüangebot im Programm **menu1.c** noch gar nicht einzulösen ist, wird bei der Wahl dieses Punktes eine Mitteilung ausgegeben, wobei eine Message-Box benutzt wird. Diese gehören nicht zu den Ressourcen, ihr Aussehen und ihre Reaktion sind vordefiniert und können beim Aufruf der Funktion **MessageBox** über die Funktions-Argumente beeinflußt werden. Die Message-Box stellt eine besonders einfach zu programmierende Alternative zu den im folgenden Abschnitt zu behandelnden Dialog-Boxen dar.

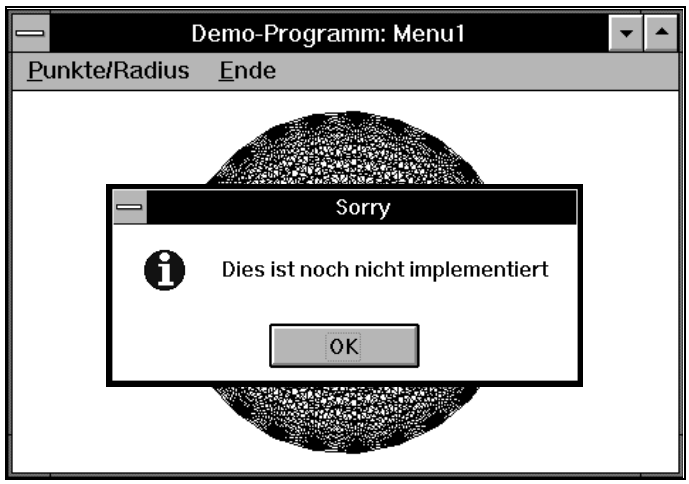
Ausschnitte aus dem Quelltext des Programms **menu1.c**:

```
/*  Menu erzeugen mit Resource-Datei (Programm menu1.c)
    ================================================== */

/*  Das Programm hat die gleiche Funktionalitaet wie das Programm rosette1.c,
    demonstriert werden zusaetzlich

    * das Definieren eines Menues ueber eine Resource-Datei und das
      Auswerten der Botschaft WM_COMMAND,

    * das Anzeigen einer einfachen Message-Box.                        */

      ...

      wndclass.lpszMenuName  = "Menu1_Menu"   ;  /* ... ist der Name des
                               Menues, das im File menu1.rc definiert ist.  */

      ...

LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)

      ...
```

```
     switch (message)
        {
          case WM_COMMAND:    /* ... wird beim Auswaehlen eines Menueangebots
                                 an das Fenster, dem das Menue zugeordnet
                                 ist, geschickt.                          */

             switch (wParam)  /* ... enthaelt die Information, welcher Menue-
                                 punkt gewaehlt wurde.                    */

                 {
                   case 10:   /* ... ist der Identifikator des Menuepunktes
                                 "Punkte/Radius", der im File menu1.rc
                                 definiert wurde.                         */

                     MessageBox (hwnd , "Dies ist noch nicht implementiert" ,
                                        "Sorry" , MB_ICONINFORMATION | MB_OK)
;
                              /* ... wird am Programm-Ende kommentiert.     */


                     return 0 ;

                   case 20:   /* ... ist der Identifikator des Menuepunktes
                                 "Ende", der im File menu1.rc definiert
                                 wurde.                                   */

                     SendMessage (hwnd , WM_CLOSE , 0 , 0L) ;
                              /* ... wird am Programm-Ende kommentiert.     */


                     return 0 ;
                 }

             break ;

    ...

/*  Meldungsfenster sind eine sehr einfache Moeglichkeit, eine Ausschrift
    auf den Bildschirm zu bringen. Mit dem Aufruf der Funktion

           MessageBox (hwnd , "Dies ist noch nicht implementiert" ,
                              "Sorry" , MB_ICONINFORMATION | MB_OK) ;

    wird ein Fenster erzeugt, das in die Titelleiste den Text "Sorry"
    schreibt. Im Fenster selbst erscheint der Text "Dies ist noch nicht
    implementiert". Das letzte Argument wird aus Bit-Flags, die in windows.h
    definiert sind, auf die uebliche Weise (mit dem "Logischen Oder")
    zusammengesetzt. Die wichtigsten Flags sind:

           MB_ICONINFORMATION  ... zeigt 'i' in einem Kreis,
           MB_ICONEXCLAMATION  ... zeigt '!' in einem Kreis,
           MB_ICONQUESTION     ... zeigt '?' in einem Kreis
           MB_ICONSTOP         ... zeigt "Stoppschild",

           MB_OK               ... zeigt "OK"-Button,
           MB_OKCANCEL         ... zeigt "OK"-Button und "Abbrechen"-Button,
           MB_YESNO            ... zeigt "Ja"-Button und "Nein"-Button,
           MB_YESNOCANCEL      ... zeigt "Ja"-Button, "Nein"-Button, und
                                         "Abbrechen"-Button,
           MB_RETRYCANCEL      ... zeigt "Wiederholen"-Button und
                                         "Abbrechen"-Button
           MB_ABORTRETRYIGNORE ... zeigt "Abbrechen"-Button, "Wiederholen"-
                                         Button und "Ignorieren"-Button

    Aus der letzten Gruppe kann sinnvollerweise jeweils nur ein Flag gesetzt
    werden, maximal 3 Buttons werden also gezeigt. Wenn ein Flag gesetzt ist,
    das mehr als einen Button erzeugt, kann alternativ eins der Flags
    MB_DEFBUTTON1, MB_DEFBUTTON2 oder MB_DEFBUTTON3 hinzugefuegt werden, das
```

```
    festlegt, welcher Button als "Default"-Button (mit dem gepunkteten
    Rechteck um die Beschriftung) gesetzt werden soll (dieser kann mit der
    Return-Taste gewaehlt werden).

    Die Message-Box eignet sich also fuer ganz einfache Dialoge (wenn der
    Benutzer z. B. nur bestaetigen oder nur "Ja" oder "Nein" sagen soll) und
    ist ganz besonders einfach zu programmieren.

    Nach der Benutzer-Reaktion wird der Eingabefokus auf das durch hwnd
    (erstes Argument) gekennzeichnete Fenster zurueckgesetzt.

    Der Return-Wert (int) korrespondiert mit den definierten Schaltflaechen
    und ist einer der in windows.h definierten Werte:

       #define IDOK         1   ** "OK"-Button gedrueckt          **
       #define IDCANCEL     2   ** "Abbrechen"-Button gedrueckt   **
       #define IDABORT      3   ** "Abbrechen"-Button gedrueckt   **
       #define IDRETRY      4   ** "Wiederholen"-Button gedrueckt **
       #define IDIGNORE     5   ** "Ignorieren"-Button gedrueckt  **
       #define IDYES        6   ** "Ja"-Button gedrueckt          **
       #define IDNO         7   ** "Nein"-Button gedrueckt        **

    Wenn die Message-Box nicht dargestellt werden kann (z. B. wegen
    mangelnden Speicherplatzes), wird der Return-Wert 0 abgeliefert.        */

/*  Mit dem Aufruf der Funktion

                    SendMessage (hwnd , WM_CLOSE , 0 , 0L) ;

    wird dem Fenster, das durch das erste Argument (hier: hwnd) gekennzeichnet
    ist, die Botschaft, die durch die drei nachfolgenden Argumente beschrieben
    wird, gesendet. Die 4 Argumente entsprechen genau den 4 Parametern, die
    die Fensterfunktion empfaengt.

    Hier schickt sich also das Hauptfenster selbst (unter Einhaltung des
    "Dienstweges") die Botschaft, dass es geschlossen werden soll. Die beiden
    letzten Parameter (wParam und lParam) haben bei dieser Botschaft keine
    Bedeutung und werden deshalb 0 gesetzt (0L ist die 'long'-0).           */

/*  Beim Arbeiten mit MS-Visual-C sollte man die Ressource-Datei menu1.rc
    zusaetzlich zu menu1.c und menu1.def in das Projekt einbeziehen. Dann
    wird automatisch im Make-File das Uebersetzen und Einbinden der Ressourcen
    vorgesehen.                                                             */
```

Die Abbildung zeigt die Reaktion auf das Wählen des Menüpunktes "Punkte/Radius". Die Message-Box ist ein "Child Window" des Hauptfensters, überlagert dieses und bekommt den Eingabefokus. Der Benutzer muß reagieren, er kann die Schaltfläche mit der Maus anklicken oder die Return-Taste drücken, weil bei nur einem Button dieser automatisch der "Default-Button" ist (tatsächlich sind auch die Buttons "Child Windows", deren "Parent Window" die Message-Box ist, die ihrerseits den Eingabefokus an einen Button weitergibt).



Message-Box überlagert Hauptfenster

Man beachte den Komfort, den Windows bei der Definition der Menüleiste über eine Ressource-Datei beisteuert: Neben der Möglichkeit, ein Menüangebot mit der Maus auszuwählen, wird auch eine komplette "Tastatur-Schnittstelle" mitgeliefert: Man kann (wie unter MS-Windows üblich) mit der Alt-Taste in das Menü wechseln, sich dort mit den Cursor-Tasten bewegen, mit der Return-Taste oder den Tasten, die durch das unterstrichene Zeichen ausgewiesen werden, ein Menü-Angebot auswählen.

10.2 Stringtable und Dialog-Box, Programm "dialog1.c"

Strings nicht "hard coded" in den Quelltext eines Programms zu schreiben, sondern in einer Ressource-Datei zu konzentrieren, ist auch dann eine gute Idee, wenn man glaubt, daß das Programm ohnehin nicht in andere Sprachen übertragen werden wird. Man findet eventuell mißverständliche Ausschriften, nicht ausreichende Fehlermeldungen usw. natürlich viel leichter (und kann sie bequem ändern), wenn sie an einer Stelle konzentriert sind.

Das Anlegen eines "Stringtables" in einer Ressource-Datei und der Zugriff auf die Strings vom Programm aus ist ganz besonders einfach und wird im Programm **dialog1.c** im Zusammenspiel mit der Datei **dialog1.rc** (gewissermaßen "nebenbei") demonstriert.

Dialog-Boxen sind für Programmierer und Anwender gleichermaßen angenehm. Bevor auch dies mit dem genannten Programm gezeigt wird, sind noch einige Erläuterungen angebracht.

10.2.1 Modale und nicht-modale Dialoge

Dialoge werden in Windows-Programmen üblicherweise in speziellen Fenstern geführt, die verschiedenartige Kontrollelemente (Knöpfe, Schaltflächen, Eingabebereiche, ...) enthalten. Art, Größe, Position und Beschriftung der Kontrollelemente werden in einer Ressource-Datei definiert ("Prototyp"-Definition der Dialog-Box).

> Für die Botschaften, die eine Dialog-Box erzeugt, ist zunächst der **Dialog-Manager** von Windows zuständig. Nur ein kleiner Teil wird an eine stets erforderliche Funktion des Anwender-Programms weitergegeben. Diese Funktion hat sehr große Ähnlichkeit mit einer "normalen" Fenster-Funktion, wird wie diese nur direkt von Windows aufgerufen und wird im folgenden als **Dialog-Funktion** bezeichnet.

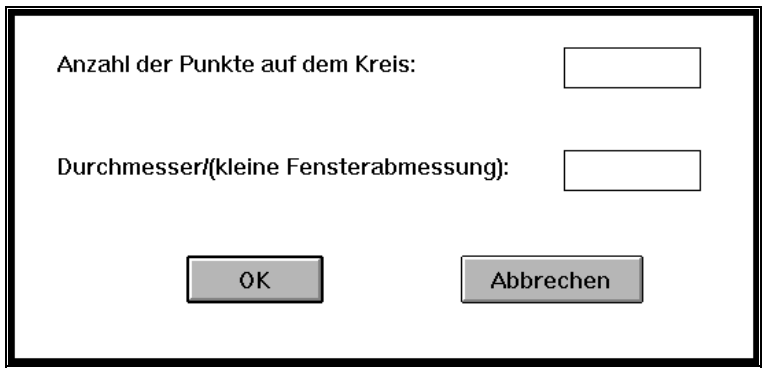
Als **modaler Dialog** wird das Arbeiten mit einer Dialog-Box bezeichnet, wenn die übrigen Fenster des Programms (also auch das Hauptfenster) erst wieder erreichbar sind, nachdem der Dialog explizit beendet wurde. Diese bevorzugte Dialogart wird auch in **dialog1.c** demonstriert. Sie sollten es einmal ausprobieren, bei aktiver Dialog-Box in das Hauptfenster zu klicken, Windows reagiert nur mit einem Piepton.

Andere Programme können dagegen auch während des modalen Dialogs aktiviert werden (es gibt allerdings auch den "systemmodalen" Dialog, bei dem außer der Reaktion auf die Dialog-Box nichts angenommen wird, "Hiermit beenden Sie Ihre Windows-Sitzung." ist ein

Beispiel für diesen speziellen Typ einer Dialog-Box, der in Anwendungs-Programmen in der Regel gar nicht verwendet wird).

Nicht-modale Dialoge gestatten auch bei geöffneter Dialog-Box das Arbeiten in anderen Fenstern des Programms. Sie erfordern einen höherem Programmieraufwand.

10.2.2 Definition einer Dialog-Box, Ressource-Datei "dialog1.rc"



Es soll die nebenstehend abgebildete einfache Dialog-Box erzeugt werden, mit der das Programm menu1.c aus dem Abschnitt 10.1 zum Programm **dialog1.c** erweitert wird. Mit diesem Programm hat der Benutzer die Möglichkeit, die Anzahl der Punkte auf dem Kreis, mit denen die Rosette gezeichnet wird, und das Verhältnis des Kreis-Durchmessers zur kleineren Fensterabmessung zu verändern. Die Dialog-Box enthält zwei unveränderliche Texte, zwei "Edit-Fenster" für die Eingabe der Zahlen und die beiden Schaltflächen für das Beenden bzw. Abbrechen der Aktion.

Die nachfolgend aufgelistete Ressource-Datei **dialog1.rc** enthält die Definition der dargestellten Dialog-Box:

```
; Ressource-Datei fuer die Definition eines einfachen Menues,
; eines "Stringtables" und einer Dialog-Box
; ===========================================================


#include <windows.h>
                      ; ... weil Konstanten-Definitionen aus
                      ; der Include-Datei verwendet werden.

Dialog1_Menu MENU
{
       MENUITEM "&Punkte/Radius"     ,  10
       MENUITEM "&Ende"              ,  20
}


; Mit dem Schluesselwort STRINGTABLE wird die Definition von String-
; Ressourcen eingeleitet. Jedem String wird ein Identifikator (UINT,
; positive ganze Zahl) vorangestellt:

STRINGTABLE
{
       1 ,      "Demo-Programm: Dialog1"
}


; Definition eines Dialog-Prototyps (die Zahlenangaben, die Positionen
; und Abmessungen beschreiben, verstehen sich in speziellen
; "Dialog-Box-Einheiten", Basis sind die Zeichenabmessungen des von
; Windows verwendeten "System-Zeichensatzes", die horizontale
```

```
; Dialog-Box-Einheit ist ein Viertel der Zeichenbreite, die vertikale
; Dialog-Box-Einheit ist ein Achtel der Zeichenhoehe):

Dialog1 DIALOG  30 , 30 , 215 , 100  ; ... definiert eine Dialog-Box,
                          ; auf die ueber den Namen "Dialog1" im
                          ; Programm Bezug genommen wird. In
                          ;
                          ;   DialogBox (hActInstance , "Dialog1" ,
                          ;              hwnd , Ptr2DialogProc)
                          ;
                          ; ist es das zweite Argument.
                          ;
                          ; 30 , 30  definiert die Position (gemessen
                          ; horizontal bzw. vertikal mit dem Ursprung
                          ; in der linken oberen Ecke), 215 ist die
                          ; horizontale, 100 die vertikale Abmessung.

STYLE    WS_POPUP | WS_DLGFRAME
                          ; ... ist der "Window-Stil", fuer den die
                          ; gleichen Konstanten aus windows.h wie
                          ; fuer das dritte Argument der Funktion
                          ; CreateWindow verwendet werden duerfen.
                          ; Die gewaehlte Kombination mit WS_POPUP
                          ; (Fenster kann an beliebiger Stelle des
                          ; Bildschirms erscheinen) und WS_DLGFRAME
                          ; (dicker Fensterrand) gilt als Standard
                          ; fuer Dialog-Boxen.

    ; Es folgt die Definition der Konstrollelemente der Dialog-Box,
    ; die einer weitgehend einheitlichen Syntax folgt:
    ;
    ;   Elementtyp  "Beschriftung" , Identifikator , x , y , b , h
    ;
    ; Fuer den Elementtyp sind nur die definierten Schluesselworte
    ; erlaubt (hier: LTEXT, EDITTEXT, ...). "Beschriftung" entfaellt
    ; fuer einige Typen (z. B. fuer EDITTEXT). Der Identifikator ist
    ; ein int-Wert, mit dem das Element im Programm identifiziert
    ; wird, x und y definieren die Position der linken oberen Ecke
    ; des Elements, b und h seine Breite bzw. Hoehe.

{
    LTEXT           "Anzahl der Punkte auf dem Kreis:" ,
                    100 , 12 , 10 , 150 , 13
                    ; ... ist ein Text, der linksbuendig in den
                    ; definierten Bereich geschrieben wird.

    EDITTEXT        110 , 160 ,  9 , 40 , 12
                    ; ... ist ein leerer Rahmen zur Aufnahme von
                    ; Text.

    LTEXT           "Durchmesser/(kleine Fensterabmessung):" ,
                    200 , 12 , 40 , 150 , 13

    EDITTEXT        210 , 160 , 39 , 40 , 12

    DEFPUSHBUTTON   "OK" , IDOK , 50 , 70 , 40 , 14
    PUSHBUTTON      "Abbrechen" , IDCANCEL , 130 , 70 , 53 , 14
                    ; ... sind rechteckige "Schaltflaechen" mit der
                    ; angegebenen Beschriftung. Als Idenfikatoren
                    ; werden die in windows.h vorgegebenen Konstanten
                    ; verwendet (vgl. Kommentar im Programm menu1.c
                    ; im Abschnitt 10.1).
                    ;
                    ; Mit DEFPUSHBUTTON wird diese Schaltflaeche
                    ; zur "Default"-Schaltflaeche erklaert (bekommt
                    ; dickeren Rahmen), die beim Betaetigen der
                    ; Return-Taste als "angeklickt" angesehen wird.
}
```

10.2.3 Quelltext des Programms "dialog1.c"

Im Quelltext des Programms **dialog1.c** wird das Zusammenspiel mit den Ressourcen, die im File **dialog1.rc** beschrieben sind, ausführlich kommentiert. Das Programm ist eine Erweiterung von menu1.c aus dem Abschnitt 10.1. Es wurde der Dialog ergänzt, und der String für die Titelleiste des Hauptfensters wird nun über die Ressource-Datei definiert.

```
/*  Arbeiten mit einer Dialog-Box (Programm dialog1.c)
    ================================================                    */

/*  Auf einem (nicht gezeichneten) Kreis werden nPoints Punkte
    gleichmaessig verteilt und jeder Punkt mit jedem anderen durch
    eine Gerade verbunden, es werden also nPoints*(nPoints-1) gerade
    Linien gezeichnet. Die Anzahl der Punkte nPoints (Voreinstel-
    lung: 25) und das Verhaeltnis von Kreisdurchmesser zur kleineren
    Fensterabmessung QuotDiamLowDist (Voreinstellung: 0.9) koennen ueber
    einen Dialog geaendert werden.

    Demonstriert werden

    * das Definieren einer Dialog-Box ueber die Resource-Datei und das
      Einrichten einer Fenster-Funktion (Dialog-Funktion) fuer diese Box,

    * die Aufrufe der Funktionen MakeProcInstance und DialogBox,

    * das Vorbelegen und Auslesen von Edit-Fenstern der Dialog-Box mit
      den Funktionen SetDlgItemText und GetDlgItemText,

    * das Definieren von Strings als Ressourcen und die Funktion
      LoadString.                                                       */

#include <windows.h>
#include <stdio.h>              /* ... fuer sprintf                    */
#include <stdlib.h>             /* ... fuer strtol und strtod          */
#include <math.h>

int      nPoints = 25  ;        /* Variablen werden global definiert, um   */
double   QuotDiamLowDist = .9 ; /* in beiden Fenster-Funktionen verfuegbar */
                                /* zu sein.                                */

/* Die beiden folgenden globalen Definitionen sind nicht zwingend, aber
   zweckmaessig:

   Beim konsequenten Arbeiten mit Strings als Ressourcen werden in der
   Regel in allen Funktionen Strings mit LoadString geladen und koennen
   so auf einem Feld "geparkt" werden, das nur einmal definiert wird.

   Einigen Funktionen (z. B. LoadString, MakeProcInstance und DialogBox) muss
   als Argument der Handle auf die aktuelle Instanz des Programms uebergeben
   werden, den WinMain beim Start als ersten Parameter erhaelt. Man kann
   diesen Handle (allerdings recht umstaendlich) auch in anderen Funktionen
   ermitteln, bequemer ist die globale Definition einer Variablen und die
   Uebergabe des Wertes in WinMain an diese Variable. Empfehlung: Das
   Windows-Skelett-Programm winskel.c, das im Abschnitt 9.4 vorgestellt
   wurde, wird um zwei Zeilen (Vereinbarung der globalen Variablen und
   Wertzuweisung in WinMain) ergaenzt.                                  */

char     WorkString [80] ;
HANDLE   hActInstance    ;   /* ... fuer den an WinMain uebergebenen Handle */

LONG FAR PASCAL WndProc    (HWND , UINT , UINT , LONG) ;
BOOL FAR PASCAL DialogProc (HWND , UINT , UINT , LONG) ;     /* ... ist der
                                                    Prototyp der Dialog-Funktion */
```

```
int PASCAL WinMain (HANDLE hInstance    , HANDLE hPrevInstance ,
                    LPSTR  lpszCmdParam , int    nCmdShow)
{
  MSG      msg      ;
  HWND     hwnd     ;
  WNDCLASS wndclass ;

  hActInstance = hInstance ;  /* ... weist den Handle auf die aktuelle
                                 Instanz der globalen Variablen zu.     */

  if (!hPrevInstance)
    {
      wndclass.style         = CS_HREDRAW | CS_VREDRAW  ;
      wndclass.lpfnWndProc   = WndProc ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
      wndclass.hInstance     = hInstance   ;
      wndclass.hIcon         = LoadIcon (NULL , IDI_APPLICATION) ;
      wndclass.hCursor       = LoadCursor (NULL , IDC_ARROW) ;
      wndclass.hbrBackground = GetStockObject (WHITE_BRUSH)  ;
      wndclass.lpszMenuName  = "Dialog1_Menu" ;
      wndclass.lpszClassName = "WndClassName" ;

      RegisterClass (&wndclass) ;
    }

  LoadString (hActInstance , 1 , WorkString , sizeof (WorkString)) ;
                  /* ... holt den String mit der Nummer 1 aus der Ressource-
                         Datei und speichert ihn auf WorkString, weitere
                         Informationen im Kommentar am Programm-Ende.      */

  hwnd = CreateWindow ("WndClassName" , WorkString ,
                 WS_OVERLAPPEDWINDOW , CW_USEDEFAULT  ,
                       CW_USEDEFAULT , CW_USEDEFAULT  , CW_USEDEFAULT ,
                         NULL , NULL , hInstance      , NULL) ;
                  /* ... verwendet den mit LoadString geholten String
                         fuer die Titelleiste des Hauptfensters.           */
  ShowWindow   (hwnd , nCmdShow) ;
  UpdateWindow (hwnd)            ;

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

      return msg.wParam ;
}



LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)
{
  HDC             hdc ;
  PAINTSTRUCT     ps  ;
  static int      cxClient , cyClient ;
  double          Radius , dPhi ;
  int             i , j , xs , ys ;
  static FARPROC  Ptr2DialogProc ;   /* ... ist ein Pointer auf eine Funktion
                    (zu diesem Datentyp existiert bereits ein Kommentar am
                    Ende des Programms miniwin.c im Abschnitt 9.3). Die
                    Variable wird static vereinbart, weil ihr beim Eintreffen
                    der Botschaft WM_CREATE ein Wert zugewiesen wird, der
                    beim Behandeln der Botschaft WM_COMMAND benutzt wird.  */

  switch (message)
        {
```

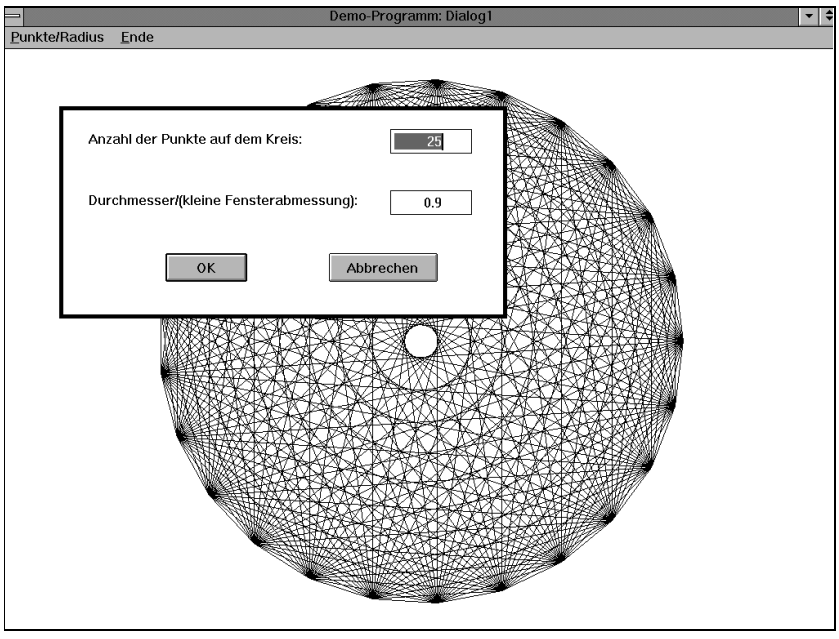
```
case WM_CREATE :  /* ... ist die Botschaft, die beim Anlegen eines
           Fensters erzeugt wird. Sie eignet sich also fuer das
           Initialisieren von Variablen, die Bezug zu diesem
           Fenster haben.                                        */

     Ptr2DialogProc = MakeProcInstance (DialogProc , hActInstance) ;
           /* ... definiert den Pointer auf die Dialog-Funktion zur
                  Abfrage von nPoints und QuotDiamLowDist.       */

     return 0 ;

  case WM_COMMAND:

   switch (wParam)
       {
         case 10:  /* ... Menue-Angebot "Punkte/Radius" gewaehlt. */

         if (DialogBox (hActInstance , "Dialog1" , hwnd ,
                        Ptr2DialogProc))
                            InvalidateRect (hwnd , NULL , TRUE) ;
           /* ... wird ausfuehrlich am Programm-Ende kommentiert. */

           return 0 ;

         case 20:  /* ... Menue-Angebot "Ende" gewaehlt.          */

           SendMessage (hwnd , WM_CLOSE , 0 , 0L) ;

           return 0 ;
       }

   break ;

case WM_SIZE :

     cxClient = LOWORD (lParam) ;
     cyClient = HIWORD (lParam) ;

     return 0 ;

case WM_PAINT :

     hdc = BeginPaint (hwnd, &ps) ;

     SetViewportOrg (hdc, cxClient / 2 , cyClient / 2) ;

     /* Die beiden globalen Variablen nPoints und QuotDiamLowDist
        wurden mit Werten vorbelegt, koennen ueber die Dialog-Box
        geaendert worden sein:                                   */


     Radius = min (cxClient , cyClient) * QuotDiamLowDist / 2 ;
     dPhi   = atan (1.) * 8. / nPoints ;

     if (Radius > 10000.) Radius = 10000. ; /* ... ist eine recht
           brutale, aber immerhin wirksame Moeglichkeit, Ueberlauf
           bei der unvermeidlichen Konvertierung auf int-Koordinaten
           zu vermeiden, bessere Moeglichkeiten findet man in
           nachfolgenden Demonstrationsprogrammen                */

     for (i = 0 ; i < nPoints ; i++)
       {
          xs = (int) (Radius * cos (dPhi * i) + .5) ;
          ys = (int) (Radius * sin (dPhi * i) + .5) ;

          for (j = 0 ; j < nPoints ; j++)
            {
               if (j != i)
```

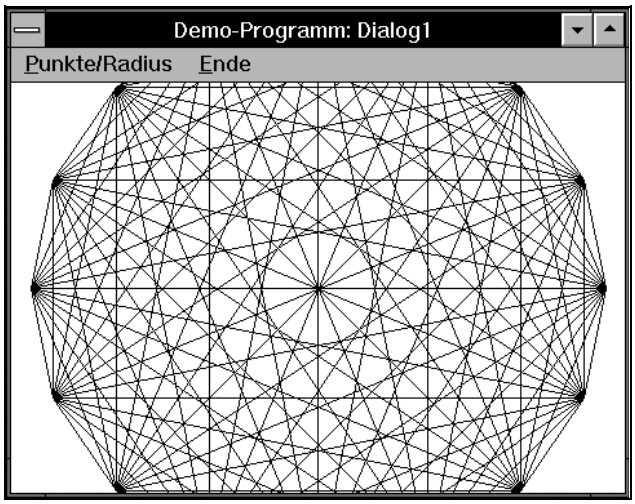
```
                        {
                           MoveTo (hdc , xs , ys) ;
                           LineTo (hdc ,
                                   (int) (Radius * cos (dPhi * j) + .5)  ,
                                   (int) (Radius * sin (dPhi * j) + .5)) ;
                        }
                     }
                  }

               EndPaint (hwnd, &ps) ;

               return 0 ;

          case WM_DESTROY :

               PostQuitMessage (0) ;

               return 0 ;
        }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}


/*  Zu einer Dialog-Box gehoert eine Fenster-Funktion (nachfolgend als
    Dialog-Funktion bezeichnet), an die Botschaften geschickt werden,
    die beim Bearbeiten der Box erzeugt werden. Eine Dialog-Funktion hat
    deshalb grosse Aehnlichkeit mit einer Fenster-Funktion, logisch
    existiert allerdings ein markanter Unterschied:

    In Windows selbst gibt es fuer die Bearbeitung von Botschaften an eine
    Dialog-Box eine eigene Fenster-Funktion, die die meisten Botschaften
    selbst bearbeitet (darin liegt der entscheidende Vorteil fuer den
    Programmierer bei der Benutzung von Dialog-Boxen), nur wenige Botschaften
    werden an die Dialog-Funktion weitergereicht. Diese gibt deshalb die
    von ihr selbst nicht bearbeiteten Botschaften auch nicht (wie eine
    "normale" Fenster-Funktion) an DefWindowProc weiter, sondern signalisiert
    diesen Tatbestand der sie aufrufenden Windows-Funktion durch den
    Return-Wert FALSE (BOOL, FALSE und TRUE sind in windows.h definiert, vgl.
    Kommentar am Ende des Programms mouse1.c im Abschnitt 9.6).

    Die vier Parameter, die eine Dialog-Funktion uebernimmt, stimmen exakt
    mit denen einer "normalen" Fenster-Funktion ueberein und charakterisieren
    die ankommende Botschaft.                                              */

BOOL FAR PASCAL DialogProc (HWND hwnd   , UINT message ,
                            UINT wParam , LONG lParam)
{
  char    *end_p ;
  int     n ;
  double  a ;

  switch (message)
  {
    case WM_INITDIALOG:   /* ... signalisiert das Erzeugen der Dialog-Box,
                             ist die erste Botschaft, die die Dialog-Funktion
                             erhaelt und entspricht der Botschaft WM_CREATE beim
                             Anlegen eines "normalen" Fensters, wird auch hier
                             fuer das Initialisieren benutzt. Die beiden EDITTEXT-
                             Felder, ueber die die Eingabe erfolgen soll, werden
                             mit den aktuellen Werten der Variablen vorbelegt:     */

      sprintf (WorkString , "%12i" , nPoints)  ;  /* ... entspricht exakt
                             der stdio-Funktion printf (formatgesteuerte Ausgabe),
                             das Ergebnis landet jedoch nicht auf dem Bildschirm,
                             sondern in der String-Variablen, die das erste Argument
                             darstellt (sprintf ist auch eine stdio-Funktion, also
                             keine Windows-Funktion)                               */
```

```
      SetDlgItemText (hwnd , 110 , WorkString) ; /* ... schreibt den in
                     WorkString enthaltenen Text in das durch den
                     Identifikator 110 gekennzeichnete Feld der durch
                     hwnd zu identifizierenden Dialog-Box.                */

      sprintf (WorkString , "%12g" , QuotDiamLowDist) ;
      SetDlgItemText (hwnd , 210 , WorkString) ;

      return TRUE ;

    case WM_COMMAND:

      switch (wParam)
        {
          case IDOK: /* ... wurde der "OK"-Button gedrueckt. Die EDITTEXT-
                                Felder werden ausgelesen:                   */

            GetDlgItemText (hwnd , 110 , WorkString , sizeof (WorkString)) ;
                     /* ... liest den im EDITTEXT-Feld (gekennzeichnet durch
                            den Idenfikator 110) der durch hwnd zu
                            identifizierenden Dialog-Box stehenden Text,legt
                            ihn auf WorkString ab, das letzte Argument
                            schuetzt vor dem Lesen eines zu langen Textes.  */

            n = (int) strtol (WorkString , &end_p , 10) ; /* ... wandelt
                            WorkString in eine long-Variable um. Die stdlib-
                            Funktion strtol wird im Kommentar des Programms
                            pointer2.c im Abschnitt 5.1 beschrieben.        */

            if (*end_p == '\0' && n > 1) nPoints = n ;

            GetDlgItemText (hwnd , 210 , WorkString , 15) ;
            a = strtod (WorkString , &end_p) ; /* strtod ist wie strtol eine
                            stdlib-Funktion und wandelt WorkString in einen
                            double-Wert um.                                 */

            if (*end_p == '\0' && a > 0.) QuotDiamLowDist = a ;

            EndDialog (hwnd , 1) ;  /* .. beendet die Arbeit der Dialog-Box
                            hwnd (Fenster wird geschlossen). Das zweite
                            Argument (int) wird von der Funktion DialogBox
                            als Return-Wert verwendet, kann also in der
                            Funktion, die DialogBox aufruft (hier: WndProc),
                            ausgewertet werden.                             */
            return TRUE ;

          case IDCANCEL:  /* ... wurde der "Abbrechen"-Button gedrueckt. Der
                                 Inhalt der EDITTEXT-Felder wird ignoriert  */

            EndDialog (hwnd , 0) ;

            return TRUE ;
        }
  }

  return FALSE ;              /* ... Botschaft wurde nicht bearbeitet.     */
}


/*  Mit dem Aufruf der Funktion

       LoadString (hActInstance , 1 , WorkString , sizeof (WorkString)) ;

    wird auf einen in einer Ressource-Datei nach dem Schluesselwort
    STRINGTABLE definierten String zugegriffen. Das erste Argument ist
    die aktuelle Instanz des Programms (wird in WinMain auf eine globale
    Variable uebertragen). Das zweite Argument gibt an, welcher String zu
    verwenden ist (alle Strings werden durch eine vorangestellte ganze
```

```
    Zahl identifiziert). In diesem Fall wird auf

                    1 ,      "Demo-Programm: Dialog1"

    zugegriffen, und der String "Demo-Programm: Dialog1" wird auf
    WorkString (drittes Argument) uebertragen. Der vierte Parameter
    schuetzt vor dem Uebertragen eines zu langen Strings.                  */

/*  Mit der Funktion DialogBox wird die Arbeit einer Dialog-Box gestartet,
    in diesem Programm aus WndProc mit

       if (DialogBox (hActInstance , "Dialog1" , hwnd , Ptr2DialogProc))
                                    InvalidateRect (hwnd , NULL , TRUE) ;

    Das erste Argument ist die aktuelle Instanz des Programms. Der Name
    "Dialog1", der als zweites Argument uebergeben wird, stellt den Bezug
    zu der in der Datei dialog1.rc mit

                               Dialog1 DIALOG ...

    definierten Dialog-Box her, hwnd ist der Handle auf das Fenster, das die
    Dialog-Box als "Child Window" erzeugt, Ptr2DialogProc ist schliesslich
    der (von WndProc beim Initialisieren festgelegte) Pointer auf die
    Dialog-Funktion (hier: DialogProc), mit dem Windows diese Funktion
    aufrufen kann.

    Als Ergebnisse des Dialogs koennen die Variablen nPoints und
    QuotDiamLowDist ihren Wert geaendert haben. Da die Dialog-Funktion von
    Windows direkt aufgerufen wird, koennen die Werte der Variablen nur durch
    ihre globale Definition an die Funktion WndProc vermittelt werden.

    Ein int-Wert wird jedoch von der Dialog-Funktion ueber EndDialog an
    Windows vermittelt und von Windows als Return-Wert von DialogBox an
    WndProc gegeben. In diesem Programm wird dieser Wert dazu verwendet,
    die Information, ob der Dialog mit "OK" oder "Abbrechen" beendet wurde,
    an WndProc "durchzureichen". Wenn eine 1 ankommt (Ende mit "OK"), wird
    eine Aenderung der Werte nPoints und QuotDiamLowDist vermutet: Das
    gesamte Fenster wird mit InvalidateRect "ungueltig" gemacht, was die
    Botschaft WM_PAINT erzeugt und damit ein Neuzeichnen veranlasst. Bei
    einem Return-Wert 0 (Ende mit "Abbrechen") wird keine Zeichenaktion
    ausgeloest (nPoints und QuotDiamLowDist haben sich nicht geaendert).   */
```

Das nebenstehende Bild zeigt das Hauptfenster des Programms, dem nach der Wahl des Menüpunktes **Punkte/Radius** die Dialog-Box überlagert wurde. In den beiden Editfenstern sind die Werte zu sehen, die als Initialisierungen für die Variablen vom Programm vorgesehen sind und die nach der Botschaft **WM_INITDIALOG** mit der Funktion **SetDlgItem-Text** in dieses Fenster geschrieben wurden.

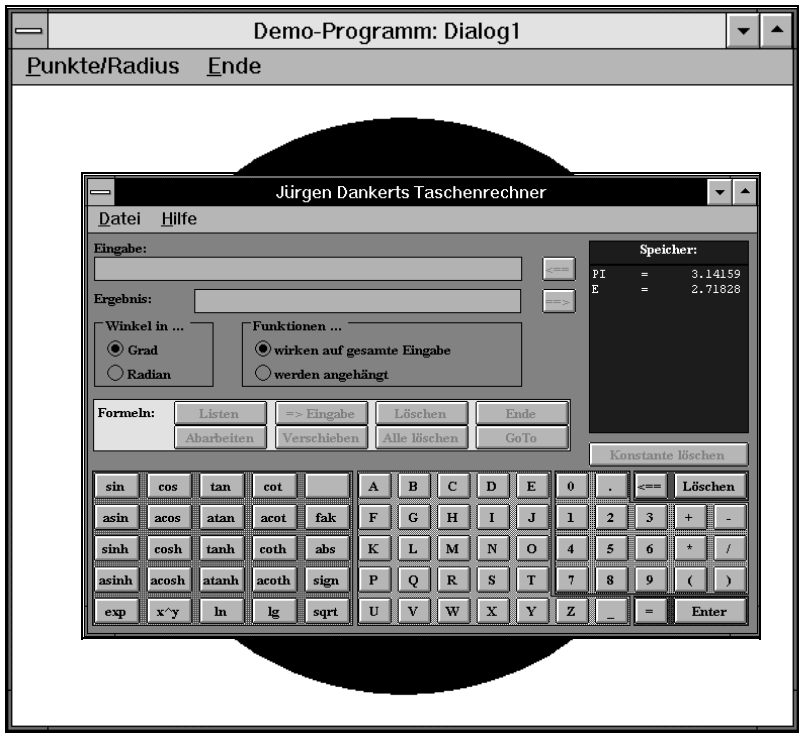




Nachdem in die Editfenster z. B. die neuen Werte **16** (Anzahl der Punkte auf dem Kreis) bzw. **1.4** (Verhältnis des Kreis-Durchmessers zur kleineren Fensterabmessung) eingegeben und die Schaltfläche "OK" angeklickt wurde, verschwindet die Dialog-Box, und das Bild wird neu gezeichnet (nebenstehende Abbildung, man beachte, daß die Zeichenfläche des Fensters als "Clipping"-Rechteck fungiert, über dessen Ränder hinaus nicht gezeichnet wird). Wenn man dagegen nach dem Ändern der Werte in den Editfenstern die Schaltfläche "Abbrechen" anklickt, verschwindet die Dialog-Box auch, aber das Bild wird nicht neu gezeichnet, der vorher verdeckte Teil des Bildes wird regeneriert.

Dies geschieht übrigens erstaunlich schnell, was man besonders dann registriert, wenn man eine größere Anzahl Punkte auf dem Kreis fordert: Bei 500 Punkten zum Beispiel braucht auch ein leistungsstarker PC für das Zeichnen etwa einer viertel Million gerader Linien einige Zeit. Aber auch für diesen Fall ist nach dem Verschwinden einer Dialog-Box durch "Abbruch" das Bild sofort regeneriert, was nur möglich ist, wenn Windows vor dem Erscheinen der Box den von ihr überdeckten Teil des Bildes sichert (Ausprobieren: 500 Punkte einstellen, "OK" wählen, nach einiger Zeit ist ein dicker schwarzer Punkt entstanden, noch einmal Dialog-Box anfordern und "Abbrechen" wählen, der dicke schwarze Punkt ist sofort repariert).

Ganz anders ist die Situation, wenn das Hauptfenster von dem Fenster eines anderen Programms überlagert wird (nebenstehendes Bild). Wenn das überlagernde Fenster verschoben oder geschlossen wird, so daß der verdeckte Teil zumindest teilweise wieder sichtbar wird, schickt Windows der zugehörigen Fenster-Funktion (in diesem Fall WndProc von dialog1.c) die Botschaft WM_PAINT und die komplette aufwendige Zeichenaktion beginnt.



Der "dicke schwarze Punkt" (249500 gerade Linien) wird vom Fenster eines anderen Programms überlagert, nach dem Verschieben dieses Fensters beginnt eine aufwendige Zeichenaktion.

Daß sich Windows um den von einer Dialog-Box verdeckten Teil des Bildschirms kümmert, ist aber nur einer von vielen Vorteilen, die mit dem Arbeiten mit dieser Variante der Eingabe von Daten verbunden ist. Alle Elemente, die zu einer Dialog-Box gehören, sind "Child Windows", die auch direkt vom Programm aus (mit CreateWindow) erzeugt werden könnten. In diesem Fall muß sich der Programmierer aber um sehr viele Dinge kümmern, die ihm der Dialog-Manager von Windows beim Arbeiten mit Dialog-Boxen abnimmt. Die aufwendigste Arbeit bei der "Selbstverwaltung" von Kontrollelementen ist das Einbeziehen der Tastatur in die Arbeit. Die "Tastatur-Schnittstelle" bekommt der Programmierer beim Erzeugen einer Dialog-Box gratis mitgeliefert:

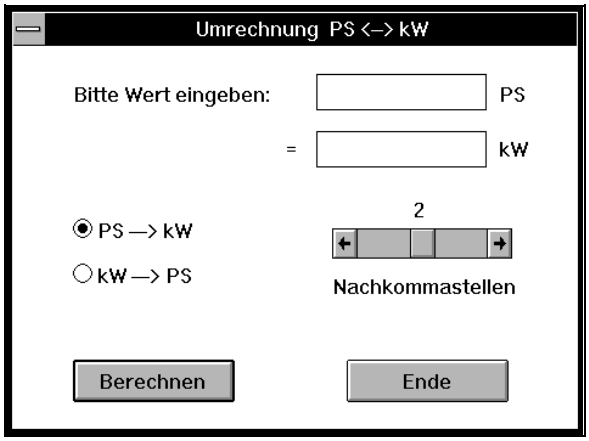
- Mit der TAB-Taste kann man von einem Kontrollelement zum nächsten wechseln (mit Shift-TAB in umgekehrter Richtung).
- Wenn der Eingabefokus auf einer Schaltfläche liegt, kann diese mit der Return-Taste oder der Leertaste ausgewählt werden.
- Wenn der Eingabefokus auf einem Editfenster liegt, wird beim Drücken der Return-Taste die Schaltfläche aktiv, die als Default-Fläche deklariert wurde (in dialog1.rc wurde die "OK"-Schaltfläche mit DEFPUSHBUTTON dafür festgelegt).
- Beim Drücken der Escape-Taste wird die Aktion abgebrochen.

10.3 Aufwendige Dialog-Boxen, Arbeiten mit einem Ressourcen-Editor

In diesem Abschnitt wird eine Windows-Version eines der ersten Programme des Tutorials ("hptokw01.c" im Abschnitt 3.5) vorgestellt. Es ist mit einer Dialog-Box ausgestattet, mit der einige neue (für das kleine Problem wohl nicht angemessene) Elemente gezeigt werden. Im nachfolgenden Abschnitt wird die Ressource-Datei "von Hand" erzeugt und das Zusammenspiel der Dialog-Box-Elemente mit der Dialog-Funktion wird ausführlich behandelt. Im Abschnitt 10.3.3 wird darauf verwiesen, daß für das Erzeugen von Ressource-Dateien sehr leistungsfähige Tools verfügbar sind.

10.3.1 Ressource-Datei "hpkwin01.rc"

Es soll die nebenstehend abgebildete Dialog-Box erzeugt werden. Über das obere "Edit-Fenster" soll der Benutzer eine Zahl eingeben, im "Edit-Fenster" darunter erscheint das Ergebnis. Die Richtung der Umrechnung (PS nach kW oder umgekehrt) soll mit den beiden (runden) "Radiobuttons" festgelegt werden können, die Anzahl der gewünschten Nachkommastellen des Ergebnisses soll mit dem "Scrollbar" (von 0 bis 4, Voreinstellung: 2) zu variieren sein. Die Dimension hinter



Eingabe- und Ergebnisfenster und die Zahl (Anzahl der Nachkommastellen) über dem "Scrollbar" soll während des Programmlaufs aktualisert werden.

Die nachfolgend gelistete Ressource-Datei **hpkwin01.rc** enthält (ausführlich kommentiert und nur deshalb so umfangreich) die Definition des Prototyps der Dialog-Box:

```
#include <windows.h>

; Der nachfolgend definierte Prototyp einer Dialog-Box zeigt,
; dass diese mit den wesentlichen Stilarten eines "normalen"
; Fensters ausgestattet werden kann (hier z. B. mit Titelleiste
; und "System-Menue").
;
; Alle Elemente einer Dialog-Box sind "Child Windows" des Dialog-
; Fensters und koennen selbst mit Fenster-Stil-Parametern
; variiert werden, die (optional) an die Definition angehaengt
; und wie ueblich durch das "Logische Oder" miteinander verknuepft
; werden koennen, hier demonstriert fuer ein EDITTEXT-Element, dem
; der Stil WS_DISABLED zugeordnet wird (im allgmeinen sind die
; Elemente per Voreinstellung mit jeweils sinnvollen Stil-
; Parametern belegt).
;
; Die mit LTEXT, CTEXT oder RTEXT (fuer die Aufnahme linksbuendigen,
; zentrierten bzw. rechtsbuendigen Textes) definierten Elemente
; gehoeren (im Gegensatz zu den mit EDITTEXT definierten Elementen)
; zur Fensterklasse "static". Dies bedeutet, dass der Benutzer den
; Text nicht aendern kann, die Dialog-Funktion kann diesen Text
; jedoch aendern, was im Programm hpkwin01.c auch genutzt wird.

HptokwinDialog DIALOG 0 , 0 , 186 , 126

STYLE      WS_POPUP | WS_CAPTION | DS_MODALFRAME | WS_SYSMENU
                          ; WS_CAPTION versieht die Dialog-Box mit
                          ; einer Titelleiste, so dass die Box
                          ; verschoben werden kann. Dieser Stil
                          ; ist automatisch mit WS_BORDER gekoppelt,
                          ; womit ein einfacher Fensterrand definiert
                          ; wird. Um zu einem kraeftigen Fensterrand
                          ; zu kommen (schliesslich soll die Dialog-
                          ; Box in diesem Programm das Hauptfenster
                          ; ersetzen), wird noch der Fensterstil
                          ; DS_MODALFRAME ergaenzt. Mit WS_SYSMENU
                          ; wird die linke obere (System-Menue-)
                          ; Schaltflaeche hinzugefuegt, die in diesem
                          ; Fall nur die Angebote "Verschieben" bzw.
                          ; "Schliessen" offerieren wird.

CAPTION  "Umrechnung  PS <--> kW"  ; ... definiert die Ueberschrift
                                   ;     fuer die Titelleiste.

{
    ; Text vor dem Eingabefeld mit LTEXT und ein leeres EDITTEXT-
    ; Fenster fuer die Eingabe, hinter dem Eingabefenster wird von
    ; der Dialog-Funktion die Dimension ("PS" bzw. "kW") in ein
    ; zunaechst leeres mit LTEXT angelegtes Feld geschrieben:
    LTEXT           "Bitte Wert eingeben:" , 200 , 20 , 12 , 75 , 13
    EDITTEXT        100 , 100 , 9 , 56 , 12
    LTEXT           " " , 220 , 160 , 12 , 20 , 8

    ; Text vor dem Ausgabefeld mit LTEXT und ein leeres EDITTEXT-
    ; Fenster fuer die Ausgabe des Ergebnisses, hinter dem
    ; Eingabefenster wird von der Dialog-Funktion die Dimension
    ; ("PS" bzw. "kW") in ein zunaechst leeres mit LTEXT angelegtes
    ; Feld geschrieben (WS_DISABLED sperrt das Fenster fuer die
    ; Eingabe):
```

```
    LTEXT           "=" , -1 , 90 , 30 , 4 , 8
    EDITTEXT        130 , 100 , 28 , 56 , 12 , WS_DISABLED
    LTEXT           " " , 230 , 160 , 30 , 20 , 8

    ; "Radiobuttons" sind runde Schaltknoepfe mit nebenstehender
    ; Beschriftung:
    RADIOBUTTON     "PS ---> kW" , 110 , 20 , 55 , 50 , 10
    RADIOBUTTON     "kW ---> PS" , 111 , 20 , 70 , 50 , 10

    ; Mit einer Laufleiste (SCROLLBAR) soll die Anzahl der
    ; gewuenschten Nachkommastellen des Ergebnisses angezeigt und
    ; und veraendert werden, mit LTEXT wird eine Beschriftung
    ; angebracht, in das zunaechst leere mit LTEXT erzeugte Feld
    ; wird von der Dialog-Funktion zusaetzlich die Anzahl der
    ; Nachkommastellen als Zahl ausgegeben:                    ;
    SCROLLBAR       120 , 105 , 60 , 59 , 10
    LTEXT           "Nachkommastellen" , 205 , 105 , 75 , 70 , 10
    LTEXT           " " , 210 , 132 , 50 , 20 , 8

    ; Der "OK"-Button soll das Berechnen ausloesen, der "Cancel"-
    ; Button das Ende des Programms:
    DEFPUSHBUTTON   "Berechnen" , IDOK , 20 , 103 , 53 , 14
    PUSHBUTTON      "Ende" , IDCANCEL , 110 , 103 , 53 , 14
}
```

Bemerkung zu den "merkwürdigen Einheiten" für Positionen und Abmessungen der Elemente einer Dialog-Box:

Windows verwendet einen "System Font", der u. a. für die Schrift in der Titelleiste von Fenstern, Menüleisten und auch Dialog-Boxen verwendet wird. Die Abmessungen ("Font Metric") der Zeichen dieses Fonts eignen sich vielfach besser für die Festlegung von Abmessungen, weil die alternative Möglichkeit (Pixel als Maßeinheit) bei unterschiedlichen Bildschirmauflösungen unpassende Größen erzeugen würde.

Da es auch bei anderen Entscheidungen über festzulegende Abmessungen vorteilhaft sein kann, sich auf die sinnvollen "System Font"-Abmessungen zu beziehen, ist sogar eine Funktion zur Abfrage dieser Werte (GetTextMetrics) verfügbar.

10.3.2 Programm "hpkwin01.c"

Das Programm "hpkwin01.c" stellt eine Besonderheit dar, die gar nicht so selten in der Windows-Programmierung zu finden ist: Das Hauptfenster des Programms bleibt unsichtbar, die Kommunikation wird ausschließlich über die Dialog-Box abgewickelt, und die Dialog-Funktion erledigt die gesamte nützliche Arbeit des Programms. Ausführlich kommentiert wird der Bezug der von der Dialog-Funktion bearbeiteten Botschaften zum Prototyp der Dialog-Box, der in der Ressource-Datei des vorigen Abschnitts definiert wurde.

```
/*  Umrechnung PS <--> kW, Programm hpkwin01.c
    ========================================                  */

/*  Dieses Programm besteht eigentlich nur aus einer Dialog-Box. Das
    Hauptfenster bleibt unsichtbar, obwohl es kreiert wird. Die Botschaft
    WM_CREATE wird sofort genutzt, um die Dialog-Box zu erzeugen. Beim Ende
    des Dialogs wird noch waehrend der Bearbeitung von WM_CREATE die
    Botschaft WM_CLOSE erzeugt, und alles ist vorbei.
```

```
    Es wird gezeigt, dass die Dialog-Box viele Eigenschaften eines
    vollwertigen Fensters annehmen kann.

    Demonstriert wird das Auswerten von Botschaften in der Dialog-Funktion,
    die von den in der Ressource-Datei definierten Elementen empfangen
    werden bzw. das Senden von Botschaften an die Elemente. Betrachtet werden

    * "Pushbuttons", die in hpkwin01.rc ueber die Schluesselworte PUSHBUTTON
      und DEFPUSHBUTTON definiert werden,

    * "Radiobuttons", die in hpkwin01.rc ueber das Schluesselwort RADIOBUTTON
      definiert werden,

    * "Statischer Text", der in hpkwin01.rc ueber das Schluesselwort LTEXT
      definiert wird,

    * "Editierbarer Text", der in hpkwin01.rc ueber das Schluesselwort
      EDITTEXT definiert wird,

    * eine "Laufleiste", die in hpkwin01.rc ueber das Schluesselwort
      SCROLLBAR definiert wird.                                          */

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char     WorkString [80] ;
HANDLE   hActInstance    ;

LONG FAR PASCAL WndProc     (HWND , UINT , UINT , LONG) ;
BOOL FAR PASCAL DialogProc  (HWND , UINT , UINT , LONG) ;
void AktualisiereStatus     (HWND , int  , int) ;

int PASCAL WinMain (HANDLE hInstance    , HANDLE hPrevInstance ,
                    LPSTR  lpszCmdParam , int    nCmdShow)
{
  MSG      msg      ;
  HWND     hwnd     ;
  WNDCLASS wndclass ;

  hActInstance = hInstance ;

  if (!hPrevInstance)
    {
      wndclass.style         = NULL  ;    /* ... ist ein bemerkenswerter
                Sonderfall, das "Fenster ohne Stil", deshalb wird von ihm
                nichts zu sehen sein. Weil es aber kreiert wird, kann es
                auch "Child Windows" haben, in diesem Programm wird es eine
                Dialog-Box sein.                                          */
      wndclass.lpfnWndProc   = WndProc ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
      wndclass.hInstance     = hInstance   ;
      wndclass.hIcon         = LoadIcon (NULL , IDI_APPLICATION) ;
      wndclass.hCursor       = LoadCursor (NULL , IDC_ARROW) ;
      wndclass.hbrBackground = GetStockObject (GRAY_BRUSH)  ;
      wndclass.lpszMenuName  = NULL ;
      wndclass.lpszClassName = "WndClassName" ;

      RegisterClass (&wndclass) ;
    }

  hwnd = CreateWindow ("WndClassName" , NULL ,
                  WS_OVERLAPPEDWINDOW , CW_USEDEFAULT  ,
                        CW_USEDEFAULT , CW_USEDEFAULT  , CW_USEDEFAULT ,
                          NULL , NULL , hInstance      , NULL) ;
```

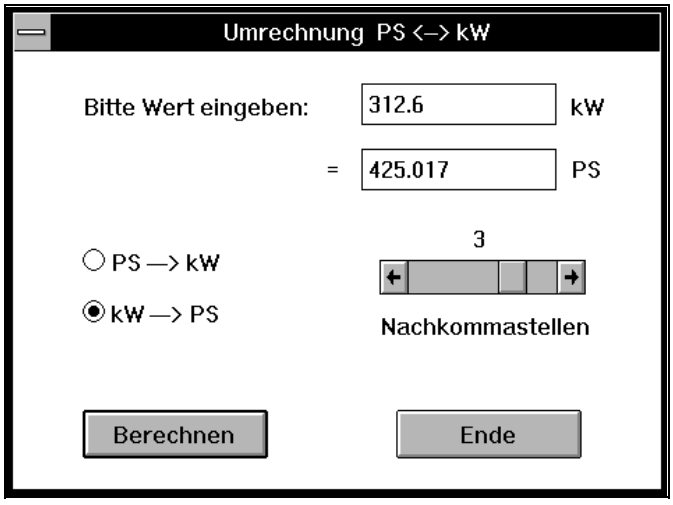
```
  /*  ShowWindow und UpdateWindow sind an dieser Stelle nicht erforderlich,
      weil das "Fenster ohne Stil" nicht gezeigt werden muss. Es kaeme hier
      ohnehin zu spaet, weil die gesamte Arbeit des Programms bereits mit
      der von CreateWindow ausgeloesten Botschaft WM_CREATE erledigt wird.  */

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

  return msg.wParam ;
}

/* Die Fenster-Funktion des Hauptfenster hat natuerlich nicht viel zu
   tun. Da das Hauptfenster ohnehin nicht zu sehen ist, wird fast alles mit
   der Botschaft WM_CREATE erledigt: Die Dialog-Box wird erzeugt, und nach
   der Beendigung des Dialogs wird sofort die Message WM_CLOSE generiert,
   und alles ist vorbei:                                                  */

LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)
{
     switch (message)
       {
         case WM_CREATE:

           DialogBox   (hActInstance , "HptokwinDialog" , hwnd ,
                        MakeProcInstance (DialogProc , hActInstance)) ;

           SendMessage (hwnd , WM_CLOSE , 0 , 0L) ;

           return 0 ;

         case WM_DESTROY:

           PostQuitMessage (0) ;
           return 0 ;
       }

     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

/* Mit der Dialog-Funktion wird demonstriert, wie die Kommunikation mit den
   Elementen einer Dialog-Box funktioniert. Diese senden Botschaften des Typs
   WM_COMMAND (Ausnahme: Laufleiste sendet Botschaft des Typs WM_HSCROLL),
   koennen abgefragt werden (z. B.: GetDlgItemText), aber es koennen ihnen
   auch Informationen geschickt werden (z. B.: SetDlgItemText).

   Die Elemente werden ueber den Handle der Dialog-Box und den Identifikator,
   der ihnen in der Ressource-Datei zugeordnet wird, angesprochen. Auch hier
   macht die Laufleiste eine Ausnahme, die ueber ihren eigenen Fenster-Handle
   identifiziert werden will (tatsaechlich sind alle Elemente einer Dialog-
   Box "Child Windows"). Mit der Funktion GetDlgItem kann der Handle eines
   Elements einer Dialog-Box ermittelt werden.

   Die nachfolgende Dialog-Funktion DialogProc verarbeitet also 3 Typen von
   Botschaften: WM_INITDIALOG kommt an, wenn die Dialog-Box geoeffnet wird,
   und WM_COMMAND bzw. WM_HSCROLL werden von den Elementen der Dialog-Box
   erzeugt:                                                               */

#define    FAKTOR    0.7355

BOOL FAR PASCAL DialogProc (HWND hwnd   , UINT message ,
                            UINT wParam , LONG lParam)
{
  static int UmrechnVariante  = 110 ;  /* ... wird mit "PS ---> kW" vorbelegt
                                                 (vgl. hpkwin01.rc)          */
```

```
  static int Nachkommastellen = 2   ;
  static int MinNachkomma     = 0   ;
  static int MaxNachkomma     = 4   ;
  double     Leistung , a           ;
  char       *end_p   ;

  switch (message)

  {
    case WM_INITDIALOG:  /* Nachricht wird zum Initialisieren genutzt:      */

      CheckRadioButton (hwnd , 110 , 111 , UmrechnVariante) ;
                        /* ... schaltet alle Optionsflaechen von Nummer
                               110 ... 111 aus und die in UmrechnVariante
                               definierte Optionsflaeche ein               */

      SetScrollRange (GetDlgItem (hwnd , 120) , SB_CTL ,
                      MinNachkomma , MaxNachkomma , FALSE) ;
                         /* GetDlgItem stellt ueber HANDLE hwnd und die
                            Kennziffer 120 den Bezug zu der definierten
                            Laufleiste her (und liefert deren Handle ab).
                            Der zweite Parameter definiert die Funktion der
                            Laufleiste (hier: SB_CTL ---> "Scroll Bar
                            Control", die haeufigsten Vertreter sind hier
                            allerdings die bekannten horizontalen bzw.
                            vertikalen Bildlaufleisten an den Fensterraendern,
                            die mit SB_HORZ bzw. SB_VERT angesprochen werden).
                            Die Parameter 3 und 4 definieren den Bereich, der
                            der gesamte Laenge der Laufleiste entspricht
                            (hier von 0 ... 4 entsprechend der Initialisierung
                            der Parameter MinNachKomma und MaxNachKomma),
                            FALSE als letzter Parameter verhindert das
                            Neuzeichnen nach dieser Definition, und ...     */

      AktualisiereStatus (hwnd , Nachkommastellen , UmrechnVariante) ;
                         /* ... aktualisiert Laufleiste und Zahlenangabe
                            ueber der Laufleiste und die Einheiten, die
                            hinter den Edit-Feldern stehen,
                            Funktion ist am Ende dieses Files definiert     */
      return TRUE ;

    case WM_HSCROLL:     /* ... ist Nachricht, die die Laufleiste ausloest  */

      switch (wParam)
        {
          case SB_LINEUP:        /* ... bei horizontalen Leisten: "LINKS"   */

            Nachkommastellen-- ;
            break ;

          case SB_LINEDOWN:      /* ... bei horizontalen Leisten: "RECHTS"  */

            Nachkommastellen++ ;
            break ;
        }

      if (Nachkommastellen > MaxNachkomma) Nachkommastellen = MaxNachkomma ;
      if (Nachkommastellen < MinNachkomma) Nachkommastellen = MinNachkomma ;

      AktualisiereStatus (hwnd , Nachkommastellen , UmrechnVariante) ;

      return TRUE ;

    case WM_COMMAND:             /* ... ist hier eine ueber die Dialog-Box
                                        abgesetzte Nachricht, ...           */
      switch (wParam)
        {
```

```
          case 110:
          case 111:

            CheckRadioButton (hwnd , 110 , 111 , (UmrechnVariante = wParam)) ;
            AktualisiereStatus (hwnd , Nachkommastellen , UmrechnVariante) ;
            SendMessage (hwnd , WM_COMMAND , IDOK , 0L) ; /* ... damit keine
                                falsche Umrechnung in den Feldern verbleibt */

            return TRUE ;

          case IDOK:     /* ... wurde die mit IDOK ("OK-Button") definierte
                                Schaltflaeche gedrueckt                    */

            GetDlgItemText  (hwnd , 100 , WorkString , 20) ;
                        /* ... liest ein "Dialog-Item" (in diesem Fall
                               einen Text, es gibt z. B. auch GetDlgItemInt)
                               aus dem durch den Schluessel 100 (vgl.
                               hpkwin01.rc) gekennzeichneten Feld in einen
                               Puffer (3. Parameter), maximal in diesem
                               Fall 20 Bytes (4. Parameter)                */

            if (strlen (WorkString) > 0)
              {
                a = strtod (WorkString , &end_p) ;
                        /* ... wandelt String in double-Variable um       */

                if (*end_p == '\0')
                  {
                    if    (UmrechnVariante == 110) Leistung = a * FAKTOR ;
                    else                           Leistung = a / FAKTOR ;

                    sprintf (WorkString , "%.*lf" , Nachkommastellen ,
                                                    Leistung) ;
                         /* ... schreibt double-Wert Leistung in den
                               WorkString mit einer Formatanweisung, die
                               die Variable Nachkommastellen aufnimmt     */

                    SetDlgItemText (hwnd , 130 , WorkString) ;
                         /* ... gibt schliesslich WorkString in das durch
                               Schluessel 130 gekennzeichnete Ergebnisfeld
                               aus                                        */
                  }
                else
                    MessageBeep (0) ;        /* ... gibt einen Piepton aus  */
              }

            return TRUE ;

          case IDCANCEL:     /* ... wurde die mit IDCANCEL ("Cancel-Button")
                                    definierte Schaltflaeche gedrueckt     */
            EndDialog (hwnd , 0) ;

            return TRUE ;
        }
  }

  return FALSE ;
}

void AktualisiereStatus (HWND hwnd , int Nachkommastellen ,
                                     int UmrechnVariante)
{
      SetScrollPos (GetDlgItem (hwnd , 120) , SB_CTL ,
                    Nachkommastellen , TRUE) ;
                         /* GetDlgItem stellt ueber HANDLE hwnd und die
                            Kennziffer 120 den Bezug zu der definierten
                            Laufleiste her (und liefert deren Handle ab).
                            Der zweite Parameter definiert (wie bei
```

```
                              SetScrollRange) die Funktion der Laufleiste
                              (hier: SB_CTL ---> "Scroll Bar Control"), der
                              dritte Parameter setzt den aktuellen Wert, auf
                              den die Marke innerhalb des mit SetScrollRange
                              definierten Bereichs gesetzt werden soll, der
                              letzte Parameter sorgt in diesem Fall fuer das
                              Neuzeichnen.                                  */

      sprintf       (WorkString , "%d" , Nachkommastellen) ;
                         /* ... wandelt Zahl in String formatgesteuert um   */

      SetDlgItemText (hwnd , 210 , WorkString) ;
                         /* ... stellt ueber HANDLE Dialog und Kennziffer
                                210 den Bezug zum Textfeld her, in das der
                                String geschrieben wird.                    */

      if (UmrechnVariante == 110)
        {
         SetDlgItemText (hwnd , 220 , "PS")     ;
         SetDlgItemText (hwnd , 230 , "kW")     ;
        }
      else
        {
         SetDlgItemText (hwnd , 220 , "kW")     ;
         SetDlgItemText (hwnd , 230 , "PS")     ;
        }
                          /* ... aktualisiert die "Dimensionen" hinter den
                                 Edit-Fenstern.                             */
      return ;
}
```

Nebenstehend abgebildet ist die Dialog-Box



- nach dem Umschalten der Berechnungsvariante (durch Anklicken des unteren "Radio-buttons") auf "kW --> PS"
- und der Änderung der gewünschten Nachkommastellen (durch Klicken auf den rechten Pfeil im "Scrollbar"),
- der Eingabe einer Zahl in das obere "Edit-Fenster"
- und dem Start der Berechnung durch Abschließen der Eingabe mit der Return-Taste oder das Anklicken des "Berechnen"-Buttons.

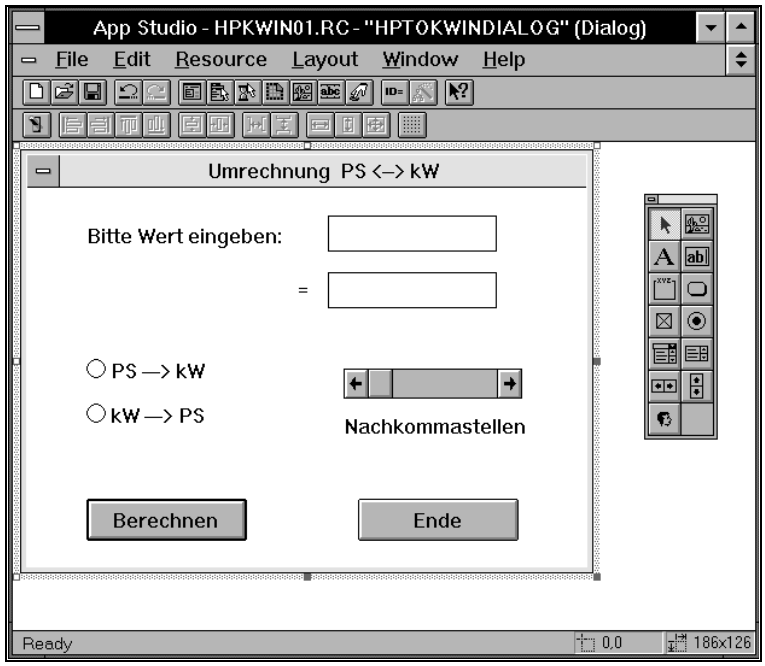
Mehr als die dargestellte Dialog-Box ist vom Programm nicht zu sehen. Dafür ist diese mit einigen Attributen eines "normalen" Fensters ausgestattet worden: Die (mit CAPTION in der Prototyp-Definition erzeugte) Kopfleiste gestattet das Verschieben der Dialog-Box auf dem Bildschirm, und das (mit "Window-Stil-Attribut" WS_SYSMENU veranlaßte) Kästchen in der linken oberen Ecke gestattet die Anforderung eines "System-Menüs".

Natürlich wird auch hier eine "Tastatur-Schnittstelle" von Windows gratis geliefert: TAB-, Alt-, Cursor-, Return- und Esc-Tasten funktionieren wie üblich. Die Esc-Taste führt hier natürlich nicht nur zum Schließen der Dialog-Box, sondern beendet auch das Programm.

10.3.3 Erzeugen eines Dialog-Prototyps mit einem Ressourcen-Editor

Das Erzeugen eines Prototyps einer Dialog-Box kann mühsam sein, weil u. a. alle Positionen und Abmessungen eingetragen werden müssen. Außerdem sind die Identifikatoren festzulegen, mit denen die Elemente in der Dialog-Funktion angesprochen werden können (der Programmierer ist also für die eindeutige Zuordnung verantwortlich). **Ressourcen-Editoren** nehmen dem Programmierer einen großen Teil der Arbeit ab. Hier soll zunächst nur auf die besonders wichtige Möglichkeit hingewiesen werden, Dialog-Boxen mit einem solchen Werkzeug zu definieren (auch alle übrigen Ressourcen können damit bearbeitet werden).

Die nebenstehende Abbildung zeigt den Ressourcen-Editor "App Studio", der zu MS-Visual-C^{++} gehört. Erstellt wird gerade der Prototyp der Dialog-Box, die im Abschnitt 10.3.1 beschrieben wurde (dort "von Hand" erzeugt).

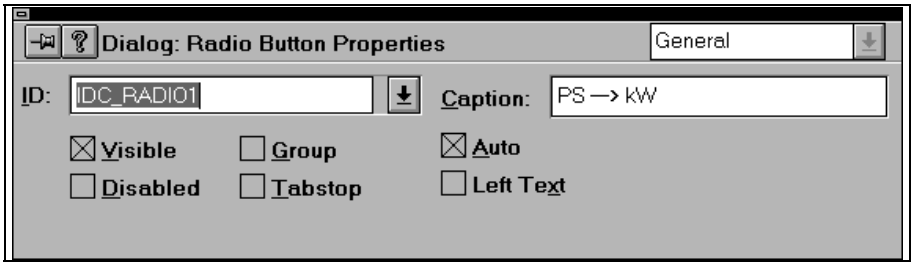


Microsofts "App Studio", ein sehr nützliches Werkzeug

Man erkennt die komfortable WYSIWYG-Eigenschaft des Editors ("What You See Is What You Get"): Aus der in der Abbildung rechts zu sehenden "App Studio Control Palette" wählt man mit der Maus das gewünschte Element aus und plaziert es ("Drag and Drop") an einer beliebigen Stelle in der Dialog-Box. Die Elemente können dort verkleinert, vergrößert und verschoben werden (auch die Größe der gesamten Dialog-Box kann natürlich geändert werden).

Für das Anpassen der zu den Elementen gehörenden Texte und der Identifikatoren dient ein spezieller Dialog, der z. B. durch Doppelklick auf ein bereits in der Dialog-Box plaziertes Element ausgelöst wird (nachfolgende Darstellung zeigt den Dialog nach Doppelklick auf den oberen "Radiobutton").

Der Text "PS --> kW" wurde bereits eingetragen. Im linken Feld ist der von "App Studio" vorgeschlagene Identfikator **IDC_RADIO1** zu erkennen:



Diesen Identifikatoren werden von "App Studio" eindeutige Werte zugewiesen, die in einer Datei **resource.h** (mit #define-Anweisungen) den Identifikatoren zugeordnet werden. Wenn diese Datei, die in die von "App Studio" erzeugte *.rc-Datei eingebunden wird, auch in das Anwenderprogramm inkludiert wird, kann der Bezug auf die Identifikatoren über diese Namen hergestellt werden.

Auf das Arbeiten mit Ressourcen-Editoren kann hier nicht weiter eingegangen werden, das Arbeiten mit "App Studio" ist durch die Menüangebote ohnehin weitgehend selbsterklärend. Nachfolgend wird zum Vergleich die mit "App Studio" erzeugte Ressource-Datei **hpkwapps.rc** für den Prototyp der Dialog-Box angegeben, für die im Abschnitt 10.3.1 die Datei **hpkwin01.rc** "von Hand" erstellt wurde. Um diese Datei ohne Änderungen für das Programm **hpkwin01.c** (Abschnitt 10.3.2) verwenden zu können, wurden für alle Identifikatoren die gleichen Zahlenwerte verwendet, die auch in der Datei hpkwin01.rc gewählt wurden (dieser Aufwand ist natürlich sonst nicht gerechtfertigt).

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

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////
//
// Dialog
//

HPTOKWINDIALOG DIALOG DISCARDABLE  0, 0, 186, 126
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Umrechnung PS <--> kW"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "Berechnen",IDOK,20,103,53,14
    PUSHBUTTON      "Ende",IDCANCEL,110,103,53,14
    SCROLLBAR       120,105,60,59,10
    LTEXT           "Nachkommastellen",205,105,75,70,10
    LTEXT           " ",210,132,50,20,8
    CONTROL         "PS ---> kW",110,"Button",BS_AUTORADIOBUTTON,20,55,50,10
    CONTROL         "kW ---> PS",111,"Button",BS_AUTORADIOBUTTON,20,70,50,10
    EDITTEXT        100,100,9,56,12,ES_AUTOHSCROLL
    EDITTEXT        130,100,28,56,12,ES_AUTOHSCROLL
    LTEXT           " ",220,160,12,20,8
    LTEXT           " ",230,160,30,20,8
    LTEXT           "Bitte Wert eingeben:",200,20,12,75,13
    LTEXT           "=",-1,90,30,4,8
END

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

////////////////////////////////////////////////////////////////////////////
///////
#endif    // APSTUDIO_INVOKED

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////////////////////////////////////////////
///////
#endif    // not APSTUDIO_INVOKED
```

Automatisch generierte Dateien sind in aller Regel ziemlich unleserlich. Die mit "App Studio" erzeugte Datei **hpkwapps.rc** ist zumindest in dem Teil, der die Definition des Prototyps der Dialog-Box enthält, ohne Schwierigkeiten interpretierbar, in einigen Passagen sogar mit der "von Hand" erstellten Datei identisch.

Auf folgende Besonderheiten der von "App Studio" erzeugten Datei soll speziell aufmerksam gemacht werden:

- Den EDITTEXT-Fenstern wird als Standard-Attribut **ES_AUTOHSCROLL** mitgegeben (dies kann natürlich in "App Studio" geändert werden). Dieses sinnvolle Attribut gestattet die Eingabe eines Textes, der eigentlich nicht "in das Fenster paßt", indem automatisch nach links oder rechts "gescrollt" wird (man stelle sich vor, diese von Windows bereitgestellte Funktionalität selbst programmieren zu wollen). Es wurde deshalb auch nicht geändert, für das Fenster, in das die Ausgabe geschrieben wird, muß dann allerdings **WS_DISABLED** weggelassen werden, denn ein langer Eingabestring kann einen langen Ausgabestring verursachen, den sich der Benutzer dann "scrollend" ansehen kann.

- Die "Radiobuttons" werden nicht mit dem Schlüsselwort **RADIOBUTTON**, sondern mit **CONTROL** definiert. Dies ist eine alternative Möglichkeit zur Definition beliebiger Elemente mit beliebigen Attributen, die man immer dann nutzen kann, wenn die vordefinierten Elemente in irgendeiner Weise nicht den Wünschen des Progammierers entsprechen (z. B., wenn man ein EDIT-Fenster ohne Rahmen haben möchte). Die Syntax für dieses generelle Format lautet:

  CONTROL "Text", ID , "Fensterklasse" , Stil , x , y , Breite , Hoehe

  mit der "Fensterklasse", für die z. B. "button", "static" (für statische Text-Fenster), "edit" (für EDITTEXT-Fenster) oder "scrollbar" gesetzt werden kann, und ganz individuell (mit dem "Logischen Oder") zusammenzustellenden Stil-Flags, die bei den vordefinierten Elementen (allerdings durchaus sinnvoll) vorgegeben sind.

10.4 Icon und Cursor

Als **Bitmaps** werden Arrays bezeichnet, mit denen einzelne Bildpunkte eines rechteckigen Bereichs beschrieben werden. Im einfachsten Fall eines monochromen Bitmaps genügt ein einzelnes Bit zur Beschreibung eines Bildpunktes (z. B.: "Schwarz oder Weiß"), bei farbigen Bildern ist die erforderliche Bit-Anzahl von der Anzahl der zugelassenen Farben abhängig (bei 16 verschiedenen Farben z. B. benötigt man 4 Bits zur Beschreibung eines Bildpunktes).

Auf Ressourcen, die durch Bitmaps beschrieben werden, wird in der Ressourcen-Datei durch die Schlüsselworte **BITMAP**, **ICON** oder **CURSOR** verwiesen. In diesen Verweisen muß ein File angegeben werden, der die eigentliche Definition enthält. In diesem Abschnitt werden selbstdefinierte Icons und Cursorformen behandelt.

10.4.1 Erzeugen von Icons und Cursorformen

Für das Erzeugen von Bitmaps benötigt man einen "Image Editor", unter MS-Visual-C^{++} übernimmt diese Arbeit das "App Studio". Die Dateien, in denen Definitionen gespeichert werden, sollten für Icons die Extension **.ico** und für Cursorformen die Extension **.cur** haben.

Sowohl Icons als auch Cursorformen werden mit 32*32 Bildpunkten dargestellt (eventuell notwendige Anpassungen für EGA-Karten werden von Windows selbst erledigt). Windows stellt sie allerdings mit einer gewissen Intelligenz dar, so daß sie bei Bildschirmen unterschiedlicher Auflösung weder unangemessen groß noch zu klein erscheinen. Während für Icons 16 Farben verwendet werden können, werden Cursor immer monochrom dargestellt.

Obwohl die Bitmaps, die Icons und Cursorformen beschreiben, immer rechteckig sind, kann durch eine spezielle Technik der Eindruck eines beliebig strukturierten Bildes erzeugt werden. Es wird nämlich neben dem eigentlichen Bitmap noch ein zweites gespeichert ("Maske" für den Hintergrund). Damit kann man beim Definieren z. B. entscheiden, ob der Hintergrund durchscheinen soll ("transparent") oder nicht ("opaque").

Die nebenstehende Abbildung zeigt das Erzeugen eines Icons mit dem "Graphics Editor" aus "App Studio" (dieses Icon wird im Programm cursor1.c im Abschnitt 10.4.3 verwendet). In der rechts zu erkennenden "Graphics Palette" werden die Farben und die gewünschten Aktionen gewählt. Die Arbeitsfläche ist groß genug, um jeden einzelnen Bildpunkt gesondert bearbeiten zu können. Links sieht man jeweils das erzeugte Bild in Originalgröße.

Erzeugen eines Icons (File fgesicht.ico) mit "App Studio"

Cursorformen werden mit dem gleichen Werkzeug und der gleichen

Technik erzeugt. Ein Cursor bekommt jedoch zusätzlich einen sogenannten "Hot Spot" zugeordnet. Dieser Punkt entspricht bei der Positionierung dem Bildschirmpunkt, der bei einer Auswahl eines Punktes mit dem Cursor (z. B. durch Mausklick) mit der entsprechenden Maus-Botschaft übergeben wird.

Die nebenstehende Abbildung zeigt die Definition eines Cursors, der im Programm cursor1.c im Abschnitt 10.4.3 verwendet wird. Als "Hot Spot" wurde die "Mitte der Nase" eingestellt.

Erzeugen eines Cursors (File tgesicht.cur) mit "App Studio"

10.4.2 Ressourcen-Datei mit Icon, Cursor, Stringtable und Menü

Die nachfolgend aufgelistete Ressourcen-Datei **cursor1.rc**, die zum Programm cursor1.c (nachfolgender Abschnitt) gehört, zeigt, wie auf die Definitionen von Icons und Cursorformen Bezug genommen wird. Den mit "App Studio" erzeugten Files fgesicht.ico, tgesicht.ico, fgesicht.cur und tgesicht.cur werden mit den Schlüsselworten **ICON** bzw. **CURSOR** die Bezeichnungen **HappyFaceIcon**, **DrearyFaceIcon**, **HappyFaceCursor** und **DrearyFaceCursor** zugeordnet, unter denen sie im Programm cursor1.c identifiziert werden.

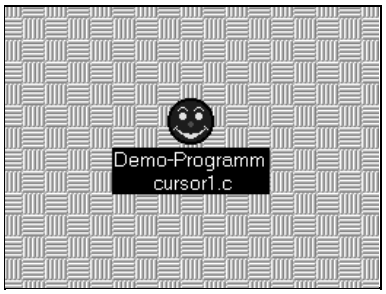
Die Definition des Menüs zeigt in Erweiterung zu den einfachen Menü-Definitionen, die in den vorangegangenen Abschnitten behandelt wurden, das Erzeugen eines "Popup-Menüs". Die Syntax dafür ist selbsterkärend:

```
; Definition von zwei Icons:

HappyFaceIcon    ICON     "fgesicht.ico"
DrearyFaceIcon   ICON     "tgesicht.ico"

; Definition von zwei Cursorformen:

HappyFaceCursor  CURSOR   "fgesicht.cur"
DrearyFaceCursor CURSOR   "tgesicht.cur"

STRINGTABLE
{
     1 ,      "Demo-Programm cursor1.c"
}

; In Menue-Definitionen sind die mit MENUITEM eingeleitenen
; Menueangebote mit Identifikatoren zu versehen, die bei den
; Botschaften WM_COMMAND, die bei der Wahl eines Menuepunktes
; gesendet werden, zur Identifizierung dienen. Im Gegensatz dazu
; folgen auf die mit POPUP eingeleiteten Menueangebote in
; Klammern die Angebote des Popup-Menues, das bei der Wahl
; dieses Menuepunktes aufgerollt wird:
```

```
Cursor1Menu MENU
{
     POPUP "&Windows-Cursor"
      {
        MENUITEM "IDC_&ARROW (Std.-Pfeil)"        ,  11
        MENUITEM "IDC_&CROSS (Kreuz)"             ,  12
        MENUITEM "IDC_&IBEAM (Text-I)"            ,  13
        MENUITEM "IDC_IC&ON (Rechteck)"           ,  14
        MENUITEM "IDC_SI&ZE (Pfeilkreuz)"         ,  15
        MENUITEM "IDC_SIZEN&ESW (Doppelpfeil /)"  ,  16
        MENUITEM "IDC_SIZE&NS (Doppelpfeil |)"    ,  17
        MENUITEM "IDC_SIZENW&SE (Doppelpfeil \\)" ,  18
        MENUITEM "IDC_SIZE&WE (Doppelpfeil --)"   ,  19
        MENUITEM "IDC_&UPARROW (Pfeil |)"         ,  20
        MENUITEM "IDC_WAI&T (Sanduhr)"            ,  21
        MENUITEM SEPARATOR     ;  ... erzeugt einen horizontalen
                               ;      Strich im Popup-Menue
        MENUITEM "&Programm-Ende"                 ,  30
      }

     MENUITEM "'&Gesichts'-Cursor"                ,  40

     POPUP "&Icon"
      {
        MENUITEM "&Froehliches Gesicht"           ,  51
        MENUITEM "&Trauriges Gesicht"             ,  52
      }

}
```

10.4.3 Programm "cursor1.c"

Das Programm **cursor1.c** greift auf die in der Ressourcen-Datei cursor1.rc (Abschnitt 10.4.2) angegebenen Ressourcen zu, speziell wird der Umgang mit Icons und verschiedenen Cursorformen demonstriert.



Programm cursor1.c "iconisiert"

Das im File fgesicht.ico definierte Icon wird dem Hauptfenster des Programms (über die wndclass-Struktur mit **RegisterClass**) zugewiesen und erscheint beim "Iconisieren" des Fensters in der nebenstehend abgebildeten Form am unteren Bildschirmrand. Der unter dem Icon stehende Text ist der mit **CreateWindow** zugewiesene Text für die Titelleiste des Fensters.

```
/*  Cursorformen und Icons (Programm cursor1.c)
    ===========================================                    */

/*  Es werden (als Icons definierte) "Traurige Gesichter" (in der linken
    Fensterhaelfte) und "Froehliche Gesichter" (in der rechten Fenster-
    haelfte) gezeichnet. Der Cursor passt bei Bewegung im Fenster seinen
    "Gesichtsausdruck" an die Gesichter im Fenster an.

    Es koennen ueber ein Menue die von Windows vordefinierten Cursor-
    formen gewaehlt werden.

    Demonstriert werden mit diesem Programm

    *  die Funktionen LoadIcon und LoadCursor,
```

```
    *  das Zuordnen eines speziellen (z. B. mit "App Studio" erzeugten
       und in der Ressourcen-Datei definierten) Icons an ein Fenster, das
       immer dann am unteren Bildschirmrand erscheint, wenn das Fenster
       "iconisiert" wird,

    *  die Moeglichkeit, dem Fenster mit SetClassWord (z. B. in Abhaengigkeit
       von der Programmsituation) ein anderes Icon zuzuweisen,

    *  das Verwenden von Icons fuer "ganz normale" Zeichenaktionen mit
       DrawIcon,

    *  das Aendern der Cursorform mit SetCursor,

    *  saemtliche in Windows vordefinierten Cursorformen und das Verwenden
       von zwei (z. B. mit "App Studio" erzeugten und in der Ressourcen-Datei
       definierten) speziellen Cursorformen,

    *  das Aendern der Cursorform in Abhaengigkeit von der aktuellen
       Cursorposition.                                                    */

#include <windows.h>

char     WorkString [80] ;
HANDLE   hActInstance    ;

LONG FAR PASCAL WndProc (HWND , UINT , UINT , LONG) ;

int PASCAL WinMain (HANDLE hInstance    , HANDLE hPrevInstance ,
                    LPSTR  lpszCmdParam , int    nCmdShow)
{
  MSG      msg      ;
  HWND     hwnd     ;
  WNDCLASS wndclass ;

  hActInstance = hInstance ;

  if (!hPrevInstance)
    {
      wndclass.style         = CS_HREDRAW | CS_VREDRAW  ;
      wndclass.lpfnWndProc   = WndProc ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
      wndclass.hInstance     = hInstance   ;
      wndclass.hIcon         = LoadIcon (hActInstance , "HappyFaceIcon") ;
                          /* ... und beim "Iconisieren" des Fensters wird das
                             in der Ressourcen-Datei cursor1.rc als

                                        HappyFaceIcon ICON ...

                             eingetragene Icon dargestellt, vgl. Kommentar
                             am Programmende.                              */
      wndclass.hCursor        = NULL ;
                          /* ... ordnet dem Fenster keinen Cursor zu, weil bei
                             jeder Botschaft WM_MOUSEMOVE ohnehin SetCursor
                             gerufen wird. Wenn die Cursorform in einem Fenster
                             geaendert werden soll, ist die Zuweisung eines
                             "NULL-Handles" an wndclass.hCursor eine gute Idee,
                             weil anderenfalls Windows bei jeder Mausbewegung
                             die Cursorform auf die wndclass.hCursor zugewie-
                             sene Form zuruecksetzt.                       */
      wndclass.hbrBackground = GetStockObject (GRAY_BRUSH)  ;
      wndclass.lpszMenuName  = "Cursor1Menu"   ;  /* ... ist der Name des
                             Menues, das im File cursor1.rc definiert ist.   */
      wndclass.lpszClassName = "WndClassName" ;

      RegisterClass (&wndclass) ;
    }
```

```
  LoadString (hActInstance , 1 , WorkString , sizeof (WorkString)) ;
                        /* ... liest den im STRINGTABLE der Datei cursor1.rc
                           mit der Nummer 1 eingetragenen String          */

  hwnd = CreateWindow ("WndClassName" , WorkString ,
                  WS_OVERLAPPEDWINDOW , CW_USEDEFAULT  ,
                        CW_USEDEFAULT , CW_USEDEFAULT  , CW_USEDEFAULT ,
                          NULL , NULL , hInstance      , NULL) ;

  ShowWindow   (hwnd , nCmdShow) ;
  UpdateWindow (hwnd)            ;

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

  return msg.wParam ;
}


LONG FAR PASCAL WndProc (HWND hwnd   , UINT message ,
                         UINT wParam , LONG lParam)
{
  static HCURSOR  hCursor ;
  static HICON    hIcond  , hIconh ;
  HDC             hdc ;
  static short    cxClient , cyClient , cxIcon , cyIcon , facecursor = 1 ;
  PAINTSTRUCT     ps ;
  short           ix , iy ;

  switch (message)
  {
    case WM_CREATE:

      /* Es soll gezeigt werden, dass die Icons auch fuer "ganz normale"
         Zeichenaktionen verwendet werden koennen. Vorbereitend werden
         sie (durch Ermitteln eines Handles) bereitgestellt, und ihre
                  Abmessungen werden ermittelt:                          */

      hIcond = LoadIcon (hActInstance , "DrearyFaceIcon") ;
      hIconh = LoadIcon (hActInstance , "HappyFaceIcon")  ;
      cxIcon = GetSystemMetrics (SM_CXICON) ;
      cyIcon = GetSystemMetrics (SM_CYICON) ;
                  /* ... und mit der Funktion GetSystemMetrics, mit der
                     die verschiedensten Informationen ueber graphische
                     Elemente zu erfragen sind, wird bei Aufruf mit den
                     Argumenten SM_CXICON bzw. SM_CYICON die horizontale
                     bzw. vertikale Abmessung (Pixel) eines Icons
                     ermittelt.                                          */
      return 0 ;

    case WM_SIZE:

      cxClient = LOWORD (lParam) ;   /* ... aktuelle Abmessungen der       */
      cyClient = HIWORD (lParam) ;   /* Zeichenflaeche des Hauptfensters   */
      return 0 ;

    case WM_PAINT:

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

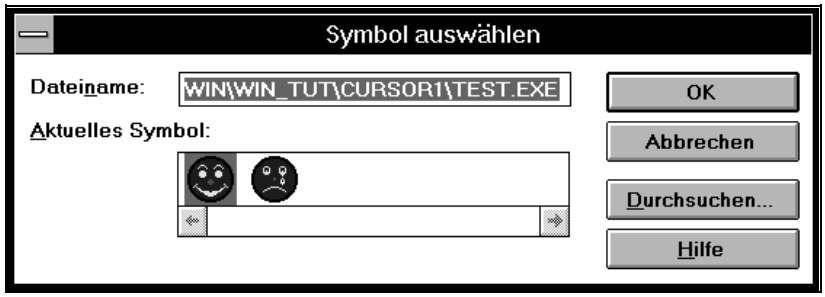
```
      /* Es werden die Icons, fuer die bei WM_CREATE die Handles ermittelt
         wurden, gezeichnet, ...                                          */

      for (iy = cyIcon/2 ; iy < cyClient ; iy += cyIcon * 3)
        {
          /* ... in der linken Fensterhaelfte "traurige Gesichter", ...     */
          for (ix = cxIcon/2 ; ix < cxClient/2 - cxIcon ; ix += cxIcon * 3)
             DrawIcon (hdc , ix , iy , hIcond) ;

          /* ... in der rechten Fensterhaelfte "froehliche Gesichter":      */
          for (ix  = cxClient - cxIcon * 3 / 2 ;
               ix >= cxClient/2 ; ix -= cxIcon * 3)
             DrawIcon (hdc , ix , iy , hIconh) ;
        }

      EndPaint (hwnd , &ps) ;

    case WM_COMMAND:                 /* ... wurde ein Menueangebot gewaehlt */

      switch (wParam)
        {
          case 11: hCursor = LoadCursor (NULL , IDC_ARROW)    ; break ;
          case 12: hCursor = LoadCursor (NULL , IDC_CROSS)    ; break ;
          case 13: hCursor = LoadCursor (NULL , IDC_IBEAM)    ; break ;
          case 14: hCursor = LoadCursor (NULL , IDC_ICON)     ; break ;
          case 15: hCursor = LoadCursor (NULL , IDC_SIZE)     ; break ;
          case 16: hCursor = LoadCursor (NULL , IDC_SIZENESW) ; break ;
          case 17: hCursor = LoadCursor (NULL , IDC_SIZENS)   ; break ;
          case 18: hCursor = LoadCursor (NULL , IDC_SIZENWSE) ; break ;
          case 19: hCursor = LoadCursor (NULL , IDC_SIZEWE)   ; break ;
          case 20: hCursor = LoadCursor (NULL , IDC_UPARROW)  ; break ;
          case 21: hCursor = LoadCursor (NULL , IDC_WAIT)     ; break ;
                   /* ... sind alle von Windows vordefinierten Cursorformen */

          case 30: SendMessage (hwnd , WM_CLOSE , 0 , 0L) ; /* Programmende */
                   return 0 ;

          case 40: facecursor = 1 ;  /* ... soll einer der beiden in
                      cursor1.rc definierten speziellen Cursor ("Trauriges"
                      oder "Froehliches Gesicht") verwendet werden.         */
                   return 0 ;

                   /* Das mit dem Aufruf von RegisterClass fuer ein Fenster
                      festgelegte Icon, das immer dann am unteren Bildschirm-
                      rand erscheint, wenn das Fenster "ikonisiert" wird,
                      kann nachtraeglich noch geaendert werden. Die Funktion
                      SetClassWord fuegt in das durch das erste Argument
                      hwnd angesprochene Fenster, wenn das zweite Argument
                      GCW_HICON ist, das als drittes Argument uebergebene
                      Icon ein (Handle auf das Icon wird hier durch
                      LoadIcon ermittelt):                                  */

          case 51: SetClassWord (hwnd , GCW_HICON ,
                                 LoadIcon (hActInstance , "HappyFaceIcon")) ;
                   return 0 ;

          case 52: SetClassWord (hwnd , GCW_HICON ,
                                 LoadIcon (hActInstance , "DrearyFaceIcon")) ;
                   return 0 ;

          default: return 0 ;
        }

       facecursor = 0 ;
       return 0 ;
```

```
    case WM_MOUSEMOVE:

        if (facecursor)  /* ... soll ein Cursor-"Gesicht" verwendet werden. */
          {
             /* Wenn sich der Cursor in der linken Fensterhaelfte befindet,
                soll es das "traurige", sonst das "froehliche" Gesicht
                sein:                                                      */
             if ((int) LOWORD (lParam) > cxClient / 2)
               hCursor = LoadCursor (hActInstance , "HappyFaceCursor") ;
             else
               hCursor = LoadCursor (hActInstance , "DrearyFaceCursor") ;
          }

        /* Erst durch das nachfolgende SetCursor (mit dem vorher durch
           LoadCursor ermittelten Handle als Argument) wird die Cursorform
           tatsaechlich gesetzt:                                           */
        SetCursor (hCursor) ;

        return 0 ;

    case WM_DESTROY:

      PostQuitMessage (0) ;
      return 0 ;
  }

  return DefWindowProc (hwnd , message , wParam , lParam) ;
}

/* Die beiden Funktionen LoadIcon und LoadCursor werden auf sehr aehnliche
   Weise verwendet. Es wird ihnen mitgeteilt, wo die Definition eines Icons
   bzw. Cursors zu finden ist, und sie liefern einen Handle auf Icon bzw.
   Cursor ab, z. B.:

            hIcond = LoadIcon (hActInstance , "DrearyFaceIcon") ;

   ... sucht im durch hActInstance zu identifizierenden Programm nach der
   Definition eines mit "DrearyFaceIcon" gekennzeichneten Icons. Es ist dort
   zu finden, weil es in der Ressourcen-Datei cursor1.rc mit

                DrearyFaceIcon    ICON     "tgesicht.ico"

   eingetragen (und in der Datei tgesicht.ico definiert) ist. Die Datei
   wurde vom Ressourcen-Compiler uebersetzt und an die EXE-Datei gebunden
   (und diese arbeitet gerade als hActInstance). Wenn das erste Argument
   (hier: hActInstance) mit dem Wert NULL belegt wird, dann vermutet
   LoadIcon ein in Windows vordefiniertes Icon, das zweite Argument muss
   in diesem Fall ein in windows.h definierter "Standard icon resource ID"
   sein:

          IDI_APPLICATION  oder  IDI_HAND         oder
          IDI_QUESTION     oder  IDI_EXCLAMATION  oder  IDI_ASTERISK

   In den Programmen der vorangegangenen Abschnitte (z. B. in dialog1.c im
   Abschnitt 10.2.3) wurde stets mit

                     LoadIcon (NULL , IDI_APPLICATION)

   ein vordefiniertes Icon dem Hauptfenster zugeordnet.

   Alle Aussagen ueber die Funktion LoadIcon gelten analog fuer die Funktion
   LoadCursor, alle vordefinierten Cursor werden mit diesem Programm
   demonstriert.                                                          */
```

Symbole im Programm-Manager

Wenn in der Ressourcen-Datei Icons verzeichnet sind, so können diese auch zur symbolischen Darstellung des Programms im Windows-Programm-Manager verwendet werden. Man "installiert" ein Programm, indem man im Programm-Manager unter dem Menüangebot "Datei" die Option "Neu" wählt. Danach muß zwischen "Programm" und "Programmgruppe" gewählt werden. Die Abbildung oben rechts zeigt eine Programmgruppe "C-Tutorial", in die drei Programme mit Icons und "Beschreibung" (wird bei der Auswahl "Programm" abgefragt) eingetragen wurden. Für die beiden Programme "Umrechnung ..." bzw. "Dialog1" (aus den Abschnitten 10.3 bzw. 10.2) mußten Icons aus dem Standard-Angebot von Windows verwendet werden, weil in den Ressourcen-Dateien dieser Programme keine Icons vorgesehen sind.

Wenn (wie beim Programm cursor1.c) in der Ressourcen-Datei Icons enthalten sind, verwendet der Programm-Manager automatisch das erste verzeichnete Icon für die symbolische Darstellung. Dies kann bei Bedarf über das Angebot "Anderes Symbol ..." geändert werden. Es erscheinen (nebenstehende Abbildung) alle verfügbaren Icons zur Auswahl.

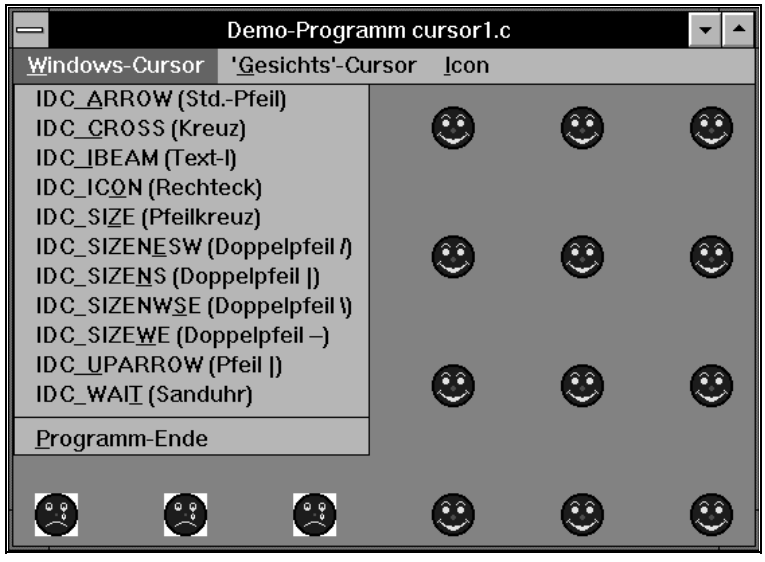


Angebot der Icons zur Auswahl als Programm-Symbol

Das Programm cursor1.c demonstriert alle in Windows vordefinierten Cursorformen, die über das Popup-Menü ausgewählt werden können. Die Abbildung unten zeigt das komplette Menü, in das die Bezeichnungen eingetragen wurden, die im File windows.h für die Cursorformen vorgesehen sind.

Die Abbildung zeigt auch, daß die bereits als Programmsymbol für den Programm-Manager und für die "Iconisierung" des Fensters benutzten Icons auch vom Programm aus gezeichnet werden können (mit der Funktion **DrawIcon**).

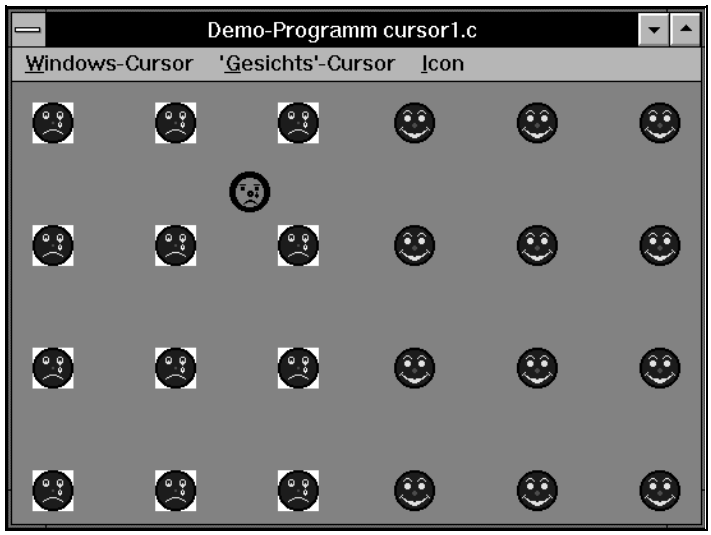
Wenn das Menüangebot 'Gesichts'-Cursor gewählt wird, erscheint einer der beiden in cursor1.rc definierten speziellen



Angebot der von Windows definierten Cursorformen

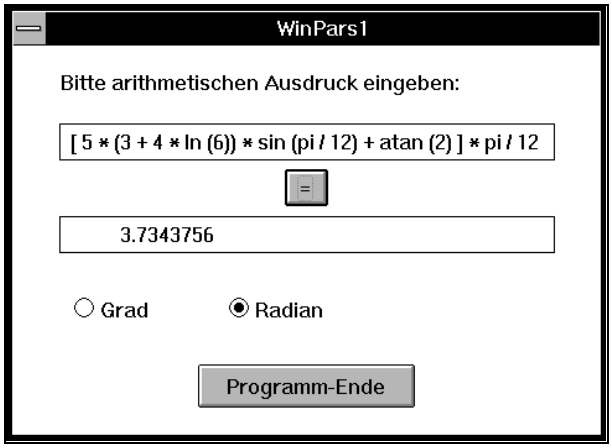
Cursor (in diesem Modus startet das Programm). Es wird die Möglichkeit demonstriert, die Cursorform auch innerhalb eines Fensters in Abhängigkeit von der Position zu ändern: Bei jeder Botschaft WM_MOUSEMOVE wird die aktuelle Cursorposition ausgewertet. Liegt sie in der linken Fensterhälfte, wird der "DrearyFaceCursor" verwendet, in der rechten Fensterhälfte erscheint der "HappyFaceCursor".

Die Abbildung zeigt auch den Unterschied zwischen den "opaque" definierten "traurigen Gesichtern" und "transparent" definierten "fröhlichen Gesichtern" und Cursorn.



Im Bereich der traurigen Gesichter ist auch der Cursor traurig, in der rechten Bildschirmhälfte wird er fröhlich

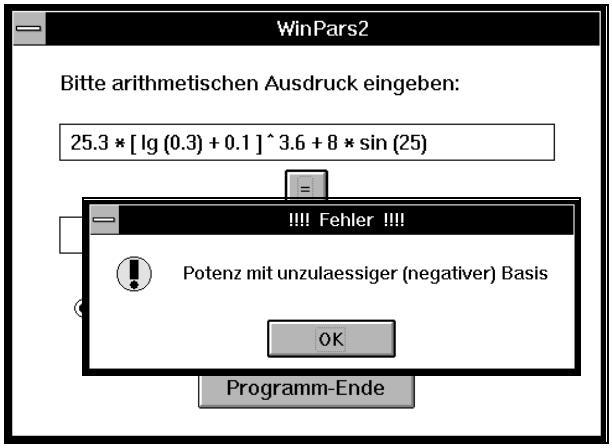
Aufgabe 10.1: Es ist ein Programm **winpars1.c** zu schreiben, das nach dem Muster des Programms hpkwin01.c (Abschnitt 10.3.2) im wesentlichen aus einer Dialog-Box besteht und in einem Edit-Fenster arithmetische Ausdrücke entgegennimmt, die mit dem mathematischen Parser (Kapitel 4) ausgewertet werden. Das Ergebnis ist in einem anderen Edit-Fenster anzuzeigen.



Die Einstellung "Grad" bzw. "Radian" für die Interpretation der Argumente der Winkelfunktionen soll über zwei "Radiobuttons" wählbar sein. Wenn bei der Auswertung des arithmetischen Ausdrucks ein Fehler erkannt wird, soll das Programm einfach "piep" sagen.

Aufgabe 10.2: Das Programm der Aufgabe 10.1 ist zu einem Programm **winpars2.c** zu verbessern:

Bei einem Fehler bei der Auswertung des arithmetischen Ausdrucks ist vom mathematischen Parser die zugehörige Fehlermeldung abzufordern. Diese ist in eine Message-Box zu schreiben, die außerdem ein in Windows vordefiniertes Icon (MB_ICONEXCLAMATION) und eine Titelleiste enthalten soll.



"Eigentlich möchte ich beim Programmieren immer alle Zusammenhänge überblicken."

"Dann solltest Du die 'Microsoft Foundation Classes' nicht benutzen."

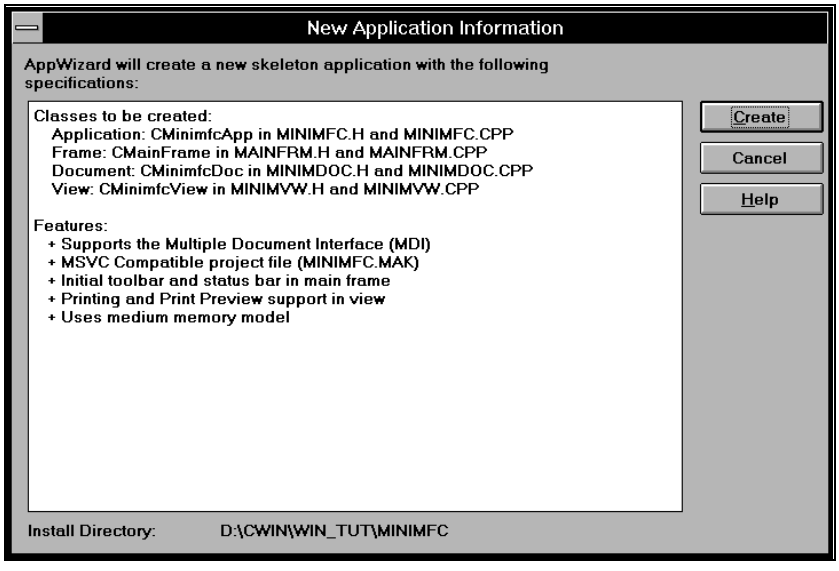
11 "Microsoft Foundation Classes" (erster Kontakt)

Beim Arbeiten mit MS-Visual-C^{++} ist die Versuchung groß, alle Angebote zu nutzen, die mit den zur Programmierumgebung gehörenden Tools offeriert werden. Als Pendant zu dem bereits in den Abschnitten 10.3.3 und 10.4 vorgestellten "App Studio" zur Bearbeitung von Ressourcen existiert der eigentliche "Magier" des Systems, der "App Wizard" zum Erzeugen von Programmen. Im Abschnitt 11.1 wird das Generieren eines kompletten Programms mit diesem Tool behandelt.

11.1 Arbeiten mit dem "App Wizard", Projekt "minimfc"

Das Programm **minimfc** wird wie das im Abschnitt 9.3 vorgestellte Programm **miniwin.c** keine Funktionalität enthalten. Es ist ein Gerüst, erzeugt vom "App Wizard", wobei alle Standard-Vorgaben akzeptiert und nicht eine einzige Programmzeile vom Programmierer ergänzt wird. Deshalb (und aus später deutlich werdenden weiteren Gründen) wird es hier nicht abgedruckt. Es wird nachfolgend das Erzeugen des Programms beschrieben:

Im Menüangebot "Project" von MS-Visual-C^{++} wird "App Wizard" gewählt. Für den darauf abgefragten "Project Name" wird z. B. **minimfc** eingegeben (im "Current Directory", das in einem Fenster angezeigt wird und gegebenenfalls geändert werden kann, wird danach vom "App Wizard" ein Subdirectory **minimfc** für die Files dieses Projektes angelegt). Alle Standard-Vorgaben werden akzeptiert (Button "OK"), und der "App Wizard" zeigt an, was er generieren wird (nebenstehende Abbildung).

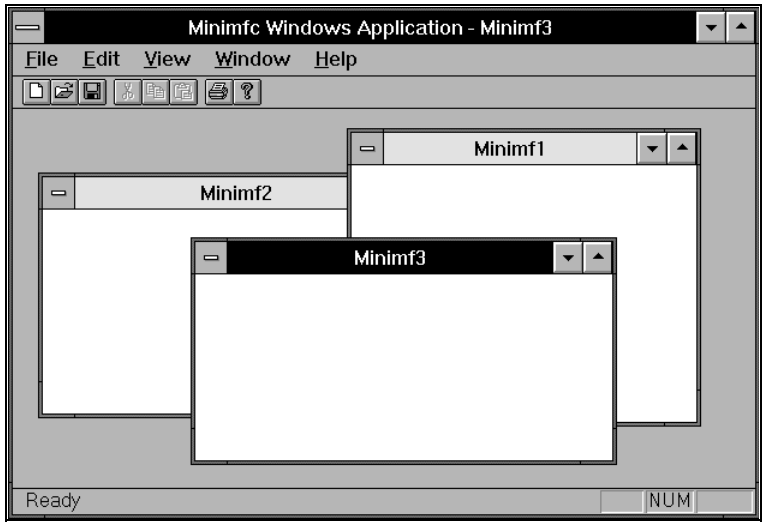


Wenn "Create" gewählt wird, beginnt die Arbeit des "App Wizard", und

nach kurzer Zeit ist das komplette Projekt erzeugt worden. Der Versuchung, sich die Files anzusehen, sollte man zunächst widerstehen (Gründe dafür später).

Compiler und Linker können ihre Arbeit beginnen, z. B.: Im Menü "Project" wird die Option "Build MINIMFC.EXE" gewählt. Man registriert, daß der Ressource-Compiler aktiv wird und daß die Programm-Dateien (es sind mehrere) die Extension **.cpp** haben.

Das Programm wird gestartet (z. B.: Option "Execute MINIMFC.EXE" im Menü "Project") und zeigt ein Hauptfenster mit geradezu üppiger Ausstattung (unter anderem Menü und Buttonleiste) und ein "Child Window". Hinter den Angeboten stecken zum Teil sogar weitere Funktionen für einen Dialog mit dem Benutzer, z. B. wird nach "File" und "Open" (oder schneller durch Anklicken des zweiten Buttons der Buttonleiste) die typische Windows-Dialogbox für das Öffnen von Dateien angeboten. Man kann auf diese Weise sogar weitere "Child Windows" erzeugen, die nebenstehende Abbildung zeigt zwei zusätzlich "Child Windows", die durch Anklicken des linken Buttons der Buttonleiste erzeugt wurden), irgendeine echte Funktionalität steckt natürlich hinter keinem der Angebote.



Programm minimfc, Hauptfenster mit drei "Child Windows"

Es ist schon beeindruckend, welchen Komfort der "App Wizard" dem Programmierer bereitstellt, ohne daß dieser auch nur eine Programmzeile geschrieben hat. Dafür muß folgender Preis gezahlt werden:

- Nach der Arbeit von "App Wizard", Compiler und Linker sind in 2 Subdirectories 37 Files mit insgesamt mehr als 3,3 MB entstanden, allein die EXE-Datei hat eine Größe von etwa 1,5 MB. Dies allerdings ist wohl das kleinere Problem.

- Der Programmierer muß nun in diesen Files die Anschlußpunkte für seine Arbeit finden, um dem Programm schließlich eine Funktionalität zu geben. Dafür gibt es selbstverständlich feste Regeln, es ist sicher nicht einfach, aber natürlich in jedem Fall effektiver als die "Eigen-Produktion" aller Funktionen, die der "App Wizard" bereits erzeugt hat.

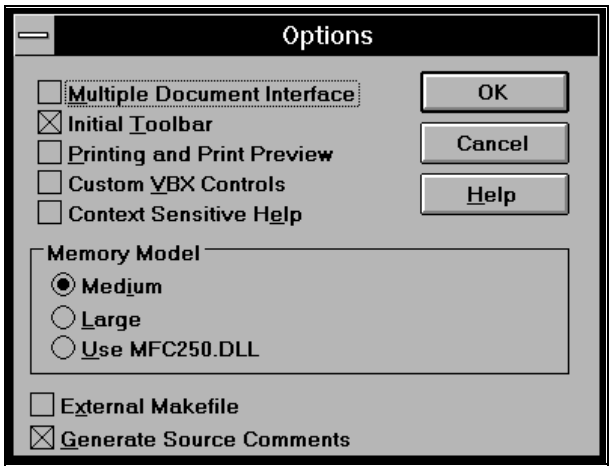
  Hinzu kommt, daß der "App Wizard" auf sehr angenehme Weise mit dem "App Studio" zusammenarbeitet. In den Abschnitten 10.3.3 und 10.4, in denen bereits mit dem "App Studio" gearbeitet wurde, ist deutlich geworden, daß eindeutige Bezüge zwischen den mit "App Studio" erzeugten Ressourcen und deren Verwendung im Programm hergestellt werden müssen. Dies läßt sich durch das Zusammenarbeiten beider Tools wesentlich sicherer gestalten.

Fazit: Der ernsthaft arbeitende Windows-Programmierer kommt an diesen Tools wohl nicht vorbei. Ein bisher noch verschwiegenes Problem (man denke an die Extensions **.cpp** der Programmdateien) kommt natürlich hinzu: Die "Microsoft Foundation Classes" (MFC), die den gesamten Komfort spendieren, verlangen die Programmierung in C++. Eine Einführung in diese Sprache wird im Kapitel 12 gegeben, nachdem im folgenden Abschnitt die "Hello-World"-Variante für MFC demonstriert wurde.

11.2 "Hello World" mit MFC, Programm "hllmfc"

Die mit den Standard-Einstellungen des "App Wizard" erzeugten Programme enthalten eine Reihe von Angeboten, die in einfachen Programmen nicht benötigt werden. Deshalb wird nach dem Start des "App Wizard" und nach dem Eingeben des Projektnamens zunächst der Button "Options" gedrückt (nebenstehende Abbildung).

Es erscheint die Dialogbox "Options". In der Darstellung unten rechts sind schon die Änderungen vorgenommen worden: "Multiple Document Interface" (Fähigkeit, Child Windows zu erzeugen) und "Printing and Print Preview" (Drucken und Druckvorschau) wurden als nicht benötigt deaktiviert. Nach Drücken des "OK"-Buttons erscheint wieder die oben dargestellte "MFC AppWizard"-Dialogbox, in der ebenfalls "OK" gewählt wird, und der "App Wizard" informiert über die zu erstellende Applikation. Nach Wahl von "Create" wird sie erzeugt.



Von den entstandenen Files haben fünf die Extension **.cpp**. Widerstehen Sie auch weiterhin der Versuchung, diese z. B. nach WinMain zu durchsuchen, um wenigstens zu erfahren, wo der Startpunkt des Programms liegt.

In eine Datei sollten Sie jedoch mit dem Editor einsteigen. In **hllmfvw.cpp** (**vw** im Namen steht für "View") findet man die "Methode OnDraw" (und dieser Begriff und der nachfolgend gelistete Code zeigen, daß es ohne C++-Kenntnisse nicht geht):

```
/////////////////////////////////////////////////////////////////////////////
// CHllmfcView drawing

void CHllmfcView::OnDraw(CDC* pDC)
{
      CHllmfcDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      // TODO: add draw code for native data here
}

/////////////////////////////////////////////////////////////////////////////
```

Die Funktion wird um zwei Anweisungen erweitert. Diese entsprechen den Anweisungen, die auch im Programm **hllwinw.c** im Abschnitt 9.5.1 bei der Bearbeitung der Botschaft WM_PAINT ausgeführt werden. Man beachte die Ähnlichkeiten, aber auch die (hier erst einmal einfach hinzunehmenden) feinen Unterschiede:

```
/////////////////////////////////////////////////////////////////////////////
// CHllmfcView drawing

void CHllmfcView::OnDraw(CDC* pDC)
{
      RECT  rect ;

      CHllmfcDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      // TODO: add draw code for native data here

      GetClientRect (&rect) ;
      pDC->DrawText ("Hello, MFC-World!" , -1 , &rect ,
                     DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
}
```

Nach dem Erzeugen des ausführbaren Programms zeigt sich, daß die Reduzierung der Optionen zumindest für den Speicherverbrauch nicht viel gebracht hat: Es existieren auch in diesem Projekt nun 37 Dateien mit insgesamt mehr als 3,2 MB, und auch die EXE-Datei ist mit etwa 1,4 MB nicht nennenswert kleiner als die EXE-Datei des Programms im vorigen Abschnitt.

Die nebenstehende Abbildung zeigt das Fenster des Programms. Ein "Child Window" existiert entsprechend der Vorgabe nicht, und der Printer-Button in der Button-Leiste ist deaktiviert.

Der Text wird nach jeder Änderung der Fenstergröße neu (zentriert) gezeichnet.

Programm hllmfc