WinFACT 98

WINDOWS FUZZY AND CONTROL TOOLS

BENUTZERHANDBUCH

USER-DLL-EXPERTE

Ingenieurbüro Dr. Kahlert
Ludwig-Erhard-Str. 45 D-59065 Hamm
Tel. 0 23 81/926 996 Fax 0 23 81/926 997

Inhalt

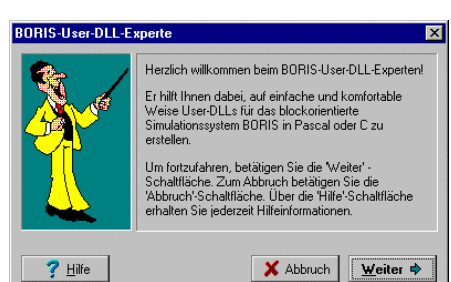
Der BORIS-User-DLL-Experte

Einführung

Übersicht

Der User-DLL-Experte unterstützt Sie bei der Entwicklung von User-DLLs für das Simulationssystem BORIS. Eine User-DLL ist ein benutzerdefinierter Systemblock in Form einer Windows-*Dynamic-Link-Library*, der zur Laufzeit des Programms automatisch geladen wird. DLLs können von nahezu allen Windows-Programmiersprachen erstellt werden. Mit Hilfe des User-DLL-Experten erstellen Sie dialoggeführt das komplette Grundgerüst der DLL in PASCAL oder C, so daß in der Regel keine oder nur noch sehr wenige Änderungen oder Ergänzungen "per Hand" im Quelltext vorgenommen werden müssen. Dieses Grundgerüst umfaßt u. a.

- Die Definition der Blockein- und -ausgänge und ihre Beschriftung
- Die Definition der Blockparameter, ihrer Default- und Grenzwerte
- Das dynamische Erzeugen und Freigeben blockspezifischer Datenstrukturen
- Die Definition der wesentlichen Simulationsfunktionen *InitSimulationDLL*, *SimulateDLL2* und *EndSimulationDLL2*



Begrüßungsdialog des User-DLL-Experten

Jeder Dialog besitzt eine Hilfe-Schaltfläche, über die Sie zugehörige Hilfeinformationen abfragen können. Folgende Hilfethemen sind zum BORIS-User-DLL-Experten verfügbar:

Inhalt

- Kommentartext
- Name und Pfad der DLL
- Ein- und Ausgänge des DLL-Blocks
- Beschriftung der Systemblockein- und -ausgänge
- Parameter des DLL-Blocks
- Vorgaben für Fließ- und Ganzzahlparameter
- Vorgaben für Schalterparameter
- Dynamische User-DLL-Daten
- Initialisierung der Simulation (Prozedur *InitSimulationDLL*)
- Simulation (Prozedur *SimulateDLL2*)
- Ende der Simulation (Prozedur *EndSimulationDLL*)
- DLL-Generierung

Jedem Hilfethema entspricht ein Abschnitt dieser Dokumentation. Darüber hinaus stehen an allgemeinen Informationen zur User-DLL-Programmierung folgende Themen im Anhang zur Verfügung, die Sie auch ausführlich im WinFACT 98-Benutzerhandbuch finden:

Anhang

- Das Konzept der User-DLLs
- Die Datenschnittstelle der User-DLL
- Anhang
- Die Funktionsschnittstelle der User-DLL
- Die Parameterstruktur *TParameterStruct*
- Datentyp *TInputArray*
- Datentyp *TOutputArray*
- Beispiele

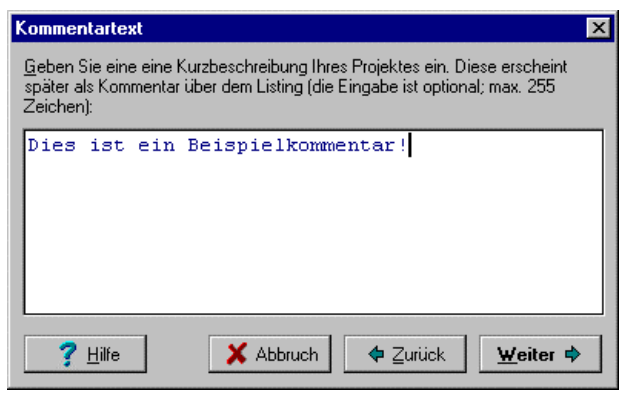
Als Vorlage (Template) für die User-DLL dienen die mitgelieferten Dateien DLLTEMPL.PAS bzw. DLLTEMPL.C. Die Vorlagedatei ist die Datei, auf deren Basis der User-DLL-Experte die User-DLL generiert, indem er an den entsprechenden Stellen die Benutzereingaben integriert. Diese Vorlagen können Sie daher bei Bedarf - z. B. zur Anpassung an spezielle Compiler - modifizieren. Dabei ist jedoch unbedingt darauf zu achten, daß die in den Originaldatei-

en eingefügten Steuersequenzen der Form "{#xyz}" nicht geändert oder entfernt werden! Die Listings beider Vorlagendateien befinden sich im Anhang.

Die vorliegende Version des User-DLL-Experten ist prädestiniert für rein funktionale DLLs ohne Visualisierungsfunktion. Eine erweiterte Version, die auch Visualisierungs-DLLs unterstützt, wird in Kürze folgen.

Kommentartext

Nachfolgender Screenshot zeigt den Dialog zur Eingabe des Kommentartextes.



Dialog zur Eingabe des Kommentartextes

Der hier eingegebene Kommentartext erscheint später im Quelltext-Kopf. Die Eingabe des Textes ist optional. Der eingegebene Text wird in der Vorlagen-Datei an der Stelle eingefügt, die durch die Steuersequenz {#COMMENT} gekennzeichnet ist.

Name und Pfad der DLL

Nachfolgender Screenshot zeigt den Dialog zur Festlegung von Name und Pfad der DLL sowie der Wahl der Programmiersprache.

Dialog zur Festlegung von Name und Pfad der DLL

Dieser Dialog erwartet folgende Eingaben:

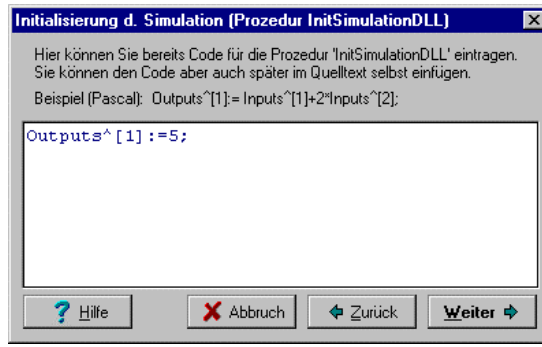
Name der DLL Dateiname, den der DLL-Quelltext erhält (ohne Dateierweiterung). Der Name darf maximal 250 Zeichen umfassen. An diesen Namen wird automatisch die Dateierweiterung DPR bzw. CPP angehängt, sofern keine andere Erweiterung angegeben wird. Der voreingestellte Name ist immer USERDLL.

Pfad der DLL Der Verzeichnispfad (Directory), in dem die DLL abgelegt wird. Dieser Pfad muß existieren!

Programmiersprache Die Programmiersprache, in der der DLL-Quelltext erzeugt wird (PASCAL oder C). Die Quelltexte sind ohne Änderungen unter Delphi 3 und C++ Builder 3 compilierbar.

Initialisierung der Simulation (Proz. *InitSimulationDLL*)

Nachfolgender Screenshot zeigt den Dialog zur Eingabe des Quelltextes für die Prozedur *InitSimulationDLL*.



Dialog zur Initialisierung der Simulation mit Beispieleingaben

Hier kann der Programmcode (d. h. der Code zwischen **begin** und **end**) für die Prozedur *InitSimulationDLL* eingefügt werden, die von BORIS vor Beginn der Simulation aufgerufen wird. Die Funktion ist wie folgt definiert:

PASCAL:

```
procedure InitSimulationDLL (D:PParameterStruct;
   Inputs:PInputArray; Outputs: POutputArray);export stdcall;
```

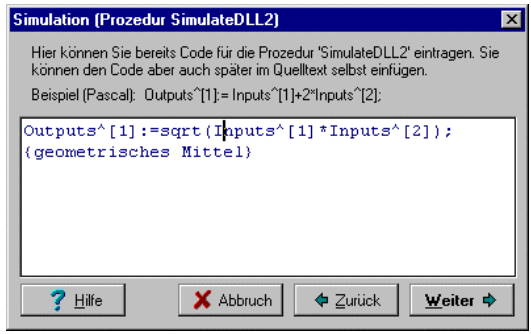
C:

```
void InitSimulationDLL (PParameterStruct  D,TInputArray  FAR
                        Inputs,TOutputArray  FAR Outputs)
```

Der eingegebene Code wird in der Vorlagen-Datei an der Stelle eingesetzt, die durch die Steuersequenz {#INIT_SIM} gekennzeichnet ist.

Simulation (Prozedur *SimulateDLL2*)

Nachfolgender Screenshot zeigt den Dialog zur Eingabe des Quelltextes für die Prozedur *SimulateDLL2*.



Dialog zur Simulation mit Beispieleingabe

Hier kann der Programmcode (d. h. der Code zwischen **begin** und **end**) für die Prozedur *SimulateDLL2* eingefügt werden, die von BORIS zu jedem Simulationsschritt aufgerufen wird. Die Funktion ist wie folgt definiert:

PASCAL:

```
procedure SimulateDLL2
  (T, DeltaT:Extended;D:PParameterStruct ;
  Inputs:PInputArray ;Outputs:POutputArray );export stdcall;
```

C:

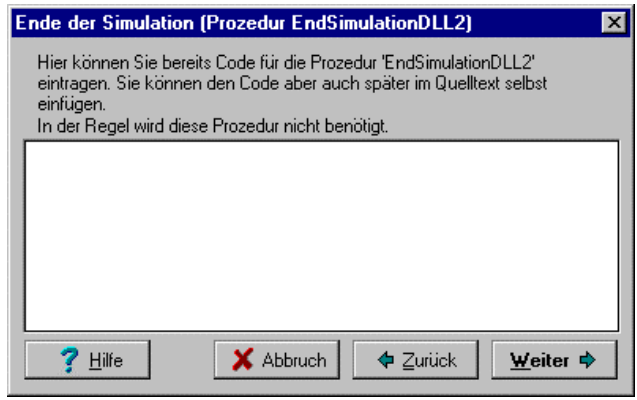
```
void SimulateDLL2
  (long double T,long double DeltaT, PParameterStruct  D,
  TInputArray  FAR Inputs,TOutputArray  FAR Outputs)
```

T ist der aktuelle Simulationszeitpunkt, *DeltaT* die Simulationsschrittweite.

Der eingegebene Code wird in der Vorlagen-Datei an der Stelle eingesetzt, die durch die Steuersequenz {#SIM} gekennzeichnet ist.

Ende der Simulation (Prozedur *EndSimulationDLL2*)

Nachfolgender Screenshot zeigt den Dialog zur Eingabe des Quelltextes für die Prozedur *EndSimulationDLL2*.



Eingabedialog für das Ende der Simulation

Hier kann der Programmcode (d. h. der Code zwischen **begin** und **end**) für die Prozedur *EndSimulationDLL2* eingefügt werden, die von BORIS nach Beendigung der Simulation aufgerufen wird. Die Funktion ist wie folgt definiert:

PASCAL:

```
procedure EndSimulationDLL2(D: PParameterStruct);
                                                export stdcall;
```

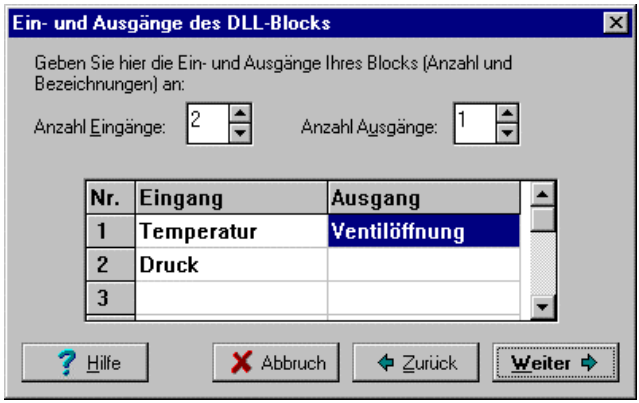
C:

```
void EndSimulationDLL2(PParameterStruct  D)
```

Der eingegebene Code wird in der Vorlagen-Datei an der Stelle eingesetzt, die durch die Steuersequenz {#END_SIM} gekennzeichnet ist.

Ein- und Ausgänge des DLL-Blocks

Nachfolgender Screenshot zeigt den Dialog zur Eingabe der Ein- und Ausgänge Ihres Blocks.

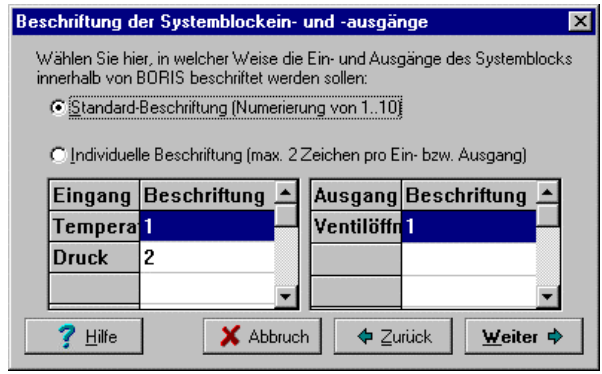


Dialog zur Eingabe der Ein- und Ausgänge des DLL-Blocks (hier nach Definition von zwei Eingängen und einem Ausgang)

In diesem Dialogfeld geben Sie an, wieviele Ein- und Ausgänge Ihr DLL-Block besitzen soll und wie diese später innerhalb des User-DLL-Parameterdialogs bezeichnet werden sollen. Ein User-DLL-Block darf jeweils maximal 50 Ein- und Ausgänge besitzen; ihre Namen müssen aus mindestens einem Zeichen bestehen und dürfen maximal 40 Zeichen aufweisen.

Beschriftung der Systemblockein- und -ausgänge

Nachfolgender Screenshot zeigt den Dialog zur Eingabe der Systemblockein- und -ausgänge.



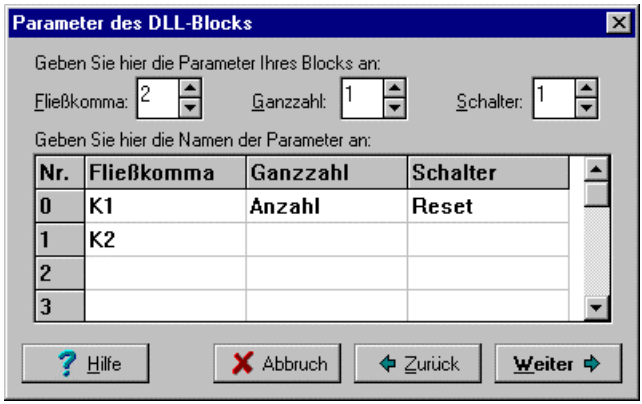
Dialog zur Beschriftung der Systemblockein- und -ausgänge

Die BORIS-User-DLL-Schnittstelle erlaubt die Vorgabe einer individuellen Beschriftung der Systemblockein- und -ausgänge (max. zwei Zeichen) innerhalb der BORIS-Systemstruktur. Standardmäßig wird bei nur einem Eingang dieser mir 'E', bei einem Ausgang dieser mit 'A' beschriftet; bei mehreren Ein- und Ausgängen werden diese fortlaufend durchnumeriert.

Wird innerhalb dieses Dialogfelds die Einstellung **Individuelle Beschriftung** gewählt, so können in den darunterliegenden Tabellen für jeden Eingang bzw. Ausgang ein oder zwei Zeichen vorgegeben werden, mit denen der Block dann später beschriftet wird (auch Leerzeichen sind möglich). Zur Beschriftung werden die Schnittstellenfunktionen *SetInputChar* bzw. *SetOutputChar* der User-DLL-Schnittstelle benutzt.

Parameter des DLL-Blocks

Nachfolgender Screenshot zeigt den Dialog zur Eingabe der Parameter des DLL-Blocks.



Dialog zur Definition der Blockparameter
(hier nach Definition von zwei Fließkommaparametern
nach je einem Ganzzahl- bzw. Schalterparameter)

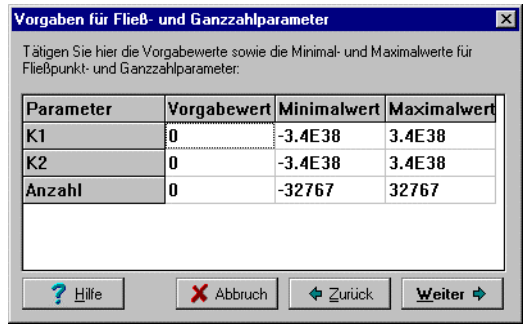
In diesem Dialog definieren Sie die Parameter Ihres Blocks. Es sind drei Typen von Parametern zu unterscheiden:

- Fließkommaparameter (Felder *E[0]..E[31]* der Parameterstruktur *TParameterStruct*)
- Ganzzahlparameter (Felder *I[0]..I[31]* der Parameterstruktur *TParameterStruct*)
- Schalterparameter (Felder *B[0]..B[31]* der Parameterstruktur *TParameterStruct*)

Ein User-DLL-Block kann von jedem Parametertyp maximal 32 Parameter besitzen; jeder Parameter muß einen Namen erhalten, der aus mindestens einem und maximal 32 Zeichen besteht.

Vorgaben für Fließ- und Ganzzahlparameter

Nachfolgender Screenshot zeigt den Dialog zur Festlegung der Vorgabewerte für Fließ- und Ganzzahlparameter.

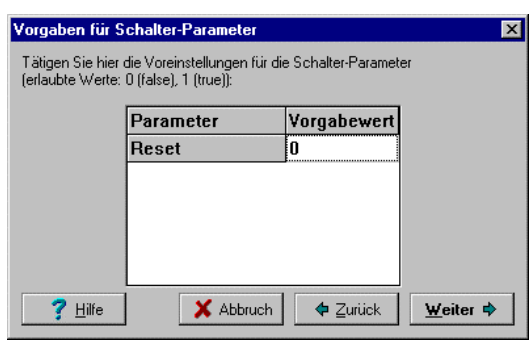


Dialog zur Festlegung der Vorgabewerte für Fließ- und Ganzzahlparameter

In diesem Dialog geben Sie die Default- und Grenzwerte für die Fließkommaparameter (Felder *E[0]..E[31]* der Parameterstruktur *TParameterStruct*) und die Ganzzahlparameter Ihres Blocks (Felder *I[0]..I[31]* der Parameterstruktur *TParameterStruct*) an.

Vorgaben für Schalterparameter

Nachfolgender Screenshot zeigt den Dialog zur Festlegung der Vorgabewerte für Schalterparameter.

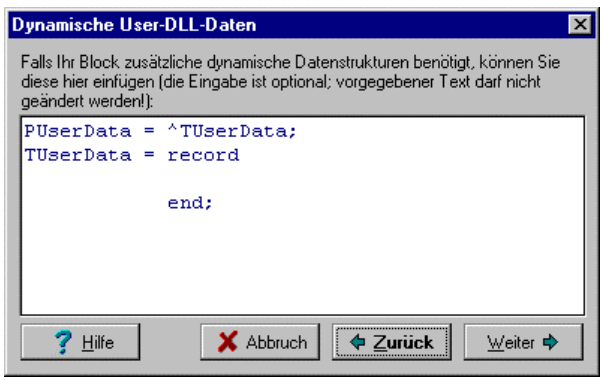


Dialog zur Festlegung der Vorgabewerte für Schalterparameter

In diesem Dialog geben Sie die Default- und Grenzwerte für die Schalterparameter Ihres Blocks (Felder *B[0]..B[31]* der Parameterstruktur *TParameterStruct*) an. Erlaubt sind die Werte 0 und 1.

Dynamische User-DLL-Daten

Nachfolgender Screenshot zeigt den Dialog zum Einfügen Dynamischer User-DLL-Daten.



Dialog zum Einfügen dynamischer Blockdaten

Bei komplexeren User-DLLs kann es erforderlich sein, dynamischen Speicher für Daten anzulegen, die Sie nicht in der Parameterstruktur *TParameterStruct* des Blocks unterbringen wollen oder können. Dies werden in der Regel z. B. Zustandsgrößen, Zwischenwerte irgendwelcher Art oder umfangreichere Parameterstrukturen sein, die beispielsweise aus einer Parameterdatei gelesen wurden. Den Zeiger auf den hier angelegten dynamischen Speicher können Sie im Parameter *UserDataPtr* von *TParameterStruct* ablegen. In diesem Dialogfeld können Sie die entsprechende Datenstruktur für Ihre dynamischen Daten definieren; diese wird dann vom User-DLL-Experten automatisch in der Prozedur *InitUserDLL* angelegt und in *DisposeUserDLL* wieder freigegeben.

Beispiel:

Es soll ein User-Block erstellt werden, der neben den Blockparametern eine 20x20-Matrix vom Typ *Extended* sowie einen Vektor der Dimension 50 vom

Typ *Integer* benötigt. Folgendes Listing zeigt die entsprechende Verwendung der Funktionen *InitUserDLL* und *DisposeUserDLL*.

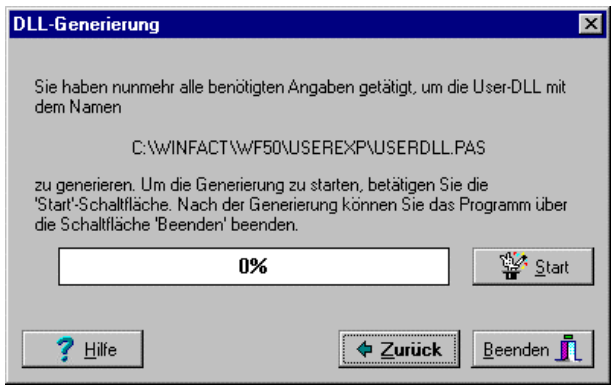
```
...
...
type
  {Deklaration des Datentyps für die User-Daten}
  PUserData = ^TUserData;
  TUserData = record
    Matrix: array[1..20, 1..20] of extended;
    Vektor: array[1..50] of integer;
  end;


procedure InitUserDLL(D: PParameterStruct); export stdcall;
begin
  ...
  ...
  {Speicher für User-Daten anfordern}
  GetMem(D^.UserDataPtr, SizeOf(TUserData));
  ...
  ...
end {of InitUserDLL};


procedure DisposeUserDLL(D: PParameterStruct); export stdcall;
begin
  ...
  ...
  {Speicher für User-Daten wieder freigeben}
  FreeMem(D^.UserDataPtr, SizeOf(TUserData));
  ...
  ...
end {of DisposeUserDLL};
...
...
```

Soll während der Simulation dann z. B. auf ein bestimmtes Matrixelement zugegriffen werden, lautet die entsprechende Pascal-Anweisung

```
...
...
{2. Zeile, 3. Spalte der Matrix auf 5 setzen}
PUserData(UserDataPtr)^.Matrix[2, 3] := 5;
...
...
```


DLL-Generierung

Nachfolgender Screenshot zeigt den Dialog zum Starten der DLL-Quelltext-Generierung.



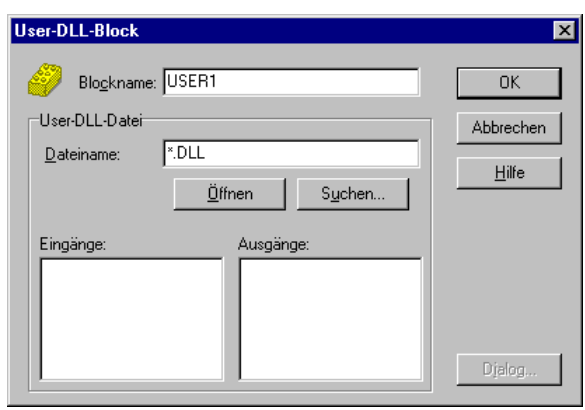
Dialog zur Generierung der DLL

Aus diesem Dialogfeld starten Sie die Generierung des DLL-Quelltextes über die 'Start'-Schaltfläche. Danach können Sie das Programm über die 'Beenden'-Schaltfläche verlassen oder auch weitere Modifikationen durchführen und dann eine erneute Quelltextgenerierung starten.

Anhang

Das Konzept der User-DLLs

BORIS erlaubt die Programmierung eigener Systemblocktypen, sog. *User-DLL-Blöcke*, auf Basis einer 32-Bit-Windows-DLL. Diese lassen sich mit praktisch jeder 32-Bit-Programmierumgebung wie z. B. Delphi 3/4, Visual Basic 5 oder Visual C++ erstellen.



Parameterdialog eines User-DLL-Blocks (hier noch leer!)

Im folgenden werden zunächst die *Datenschnittstelle* und die *Funktionsschnittstelle* des User-Blocks besprochen, bevor dann anhand von Beispielen unterschiedlicher Komplexität eine tiefergehende Einführung in die Programmierung von User-Blöcken erfolgt.

Hinweis: Alle nachfolgend abgedruckten Programme bzw. Programmsegmente wurden mit Delphi 3 erstellt. Eine Übertragung auf andere Entwicklungsumgebungen oder Programmiersprachen ist aber ohne größere Probleme möglich.

Die Datenschnittstelle des User-Blocks

Alle für den Anwender relevanten Daten eines User-Blocks sind in einer Datenstruktur vom Typ *TParameterStruct* bzw. einem Zeiger *PParameterStruct* auf diese Struktur festgelegt. Diese Datenstruktur enthält zunächst die eigentlichen *Blockparameter*. Diese Daten sind in der Regel blockspezifische Konstanten (z. B. Verstärkungsfaktoren, Zeitkonstanten oder ähnliches) und können vom Anwender über den Parameterdialog des User-Blocks geändert werden. Sie werden beim Speichern einer BORIS-Struktur mit dem User-Block abgespeichert und stehen damit nach dem erneuten Einlesen der Datei wieder zur Verfügung. Darüber hinaus können diese Parameter natürlich auch für andere Zwecke - z. B. als Zustandsvariablen, Zwischen- oder Hilfsvariablen irgendwelcher Art, Flags usw. - "mißbraucht" werden.

Folgende Parameter stehen zur Verfügung:

- Bis zu 32 Fließkommazahlen (10 Byte, Datentyp *extended* in PASCAL bzw. *long double* in C)
- Bis zu 32 Integer-Zahlen (4 Byte, Datentyp *longint* bzw. *integer* in Pascal bzw. *int* in C)
- Bis zu 32 Schalter- bzw. Byte-Variablen (1 Byte, Datentyp *byte* in Pascal bzw. *char* in C)
- Eine Stringvariable der Länge 256 (z. B. für Dateinamen)

Für die Fließkomma- bzw. Integer-Parameter können Minimal- und Maximalwerte vorgegeben werden, deren Einhaltung dann im Parameterdialog automatisch überprüft wird. Für alle Parameter können bzw. müssen außerdem Namen vergeben werden, die dann an entsprechender Stelle im Parameterdialog erscheinen. Weitere Parameter können bei Bedarf auch aus einer zusätzlichen Datei gelesen werden. Neben diesen eigentlichen Blockparametern enthält die Datenstruktur *TParameterStruct* einige weitere Variablen, die nur für fortgeschrittene Anwendungen benötigt werden und an späterer Stelle im Rahmen der

Beispiele genauer erläutert werden. Zunächst folgt eine knappe Auflistung der Komponenten von *TParameterStruct*.

```
type
  PParameterStruct = ^TParameterStruct;
  TParameterStruct = packed record
    NuE: Byte;                      {Anzahl Fließkomma-Parameter}
    NuI: Byte;                      {Anzahl Integer-Parameter}
    NuB: Byte;                      {Anzahl Schalter-Parameter}
    E: Array[0..31] of Extended;    {Fließkomma-Parameter}
    I: Array[0..31] of LongInt;     {Integer-Parameter}
    B: Array[0..31] of Byte;        {Schalter-Parameter}
    D: Array[0..255] of char;       {Dateiname für optionale Daten}
    EMin: Array[0..31] of Extended; {unt. Grenze Fließkomma-Par.}
    EMax: Array[0..31] of Extended; {obere Grenze Fließkomma-Par.}
    IMin: Array[0..31] of LongInt;  {untere Grenze Integer-Par.}
    IMax: Array[0..31] of LongInt;  {obere Grenze Integer-Par.}
    NaE: Array[0..31,0..40] of char; {Namen Fließkomma-Parameter}
    NaI: Array[0..31,0..40] of char; {Namen Integer-Parameter}
    NaB: Array[0..31,0..40] of char; {Namen Schalter-Parameter}
    UserDataPtr: Pointer;            {Zeiger auf opt. Variablen}
    ParentPtr: Pointer;              {Zeiger auf User-DLL-Block}
    ParentHWnd: Word;                {BORIS-Fensterhandle}
    ParentName: PChar;               {Name des User-DLL-Blocks}
    UserHWindow: Word;               {Benutzerdef. Fensterhandle,
                                       z. B. für Ausgabefenster}
    DataFile: text;                  {Optionale Textdatei}
  end;
```

Struktur der Datenschnittstelle TParameterStruct *in Pascal*

Variable	Bedeutung
NuE, NuI, NuB	Enthält die Anzahl der Fließkomma-, Integer- bzw. Schalterparameter des Blocks. Dabei sind nur die wirklich als Blockparameter benutzten Komponenten zu berücksichtigen, die im Parameterdialog erscheinen sollen, nicht aber die als Hilfsgrößen (z. B. Zustandsvariablen, Flags o. ä.) verwendeten Komponenten der Arrays *E*, *I* bzw. *B*!
E, I, B	Diese Arrays enthalten die Parameter selbst.
D	Enthält bei Bedarf den Namen einer externen Datei für das Einlesen weiterer Daten.
EMin, EMax	Diese Arrays enthalten die unteren bzw. oberen zulässigen Werte für die Fließkomma-Parameter
IMin, IMax	Diese Arrays enthalten die unteren bzw. oberen zulässigen Werte für die Integer-Parameter.

NaE, NaI, NaB	Arrays mit den Namen der Fließkomma-, Integer- bzw. Schaltervariablen. Jeder Name darf maximal 40 Zeichen aufweisen
UserDataPtr	Zeiger auf optionale Blockvariablen (z. B. Zustandsgrößen, Zwischenwerte oder ähnliches). Hierunter sind in der Regel solche Variablen zu verstehen, die keine Blockparameter sind und damit auch nicht mit dem Block abgespeichert werden müssen.
ParentPtr	Dieser Zeiger zeigt auf den User-DLL-Block. Wird für Anwender-DLLs im allgemeinen nicht benötigt.
ParentHWnd	Handle des BORIS-Hauptfensters. Wichtig bei User-DLLs, die eigene Fenster (z. B. zur Visualisierung) oder Dialoge besitzen.
ParentName	Zeiger auf den Namen des User-DLL-Blocks. Wird i. a. nur benötigt bei User-DLL-Blöcken mit Visualisierungsfunktion
UserHWindow	In dieser Variablen kann z. B. das Fensterhandle eines mit dem User-Block generierten Visualisierungsfensters gespeichert werden. Wird i. a. nur benötigt bei User-DLL-Blöcken mit Visualisierungsfunktion.
DataFile	Textdatei für universelle Zwecke. Kann zusammen mit der Variablen *D* (s. o.) verwendet werden.

Neben der wichtigsten Datenstruktur *TParameterStruct* müssen in der User-DLL einige weitere Datenstrukturen deklariert sein. Dazu gehört zunächst die Struktur *TDialogEnableStruct* bzw. der zugehörige Zeiger *PDialogEnableStruct*.

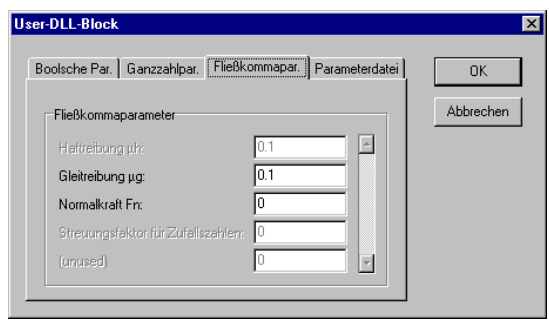
```
type
  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnableStruct=record
    AllowE: Longint;          { Soll die Eingabe eines Wertes   }
    AllowI: Longint;          { un-/zulässig sein so ist das Bit}
    AllowB: Longint;          { des Allow?-Feldes 0 bzw. 1}
    AllowD: Byte;
  end;
```

Struktur von TDialogEnableStruct *(in Pascal)*

Diese Struktur wird zur Verwaltung des Parameterdialogs benötigt; sie ermöglicht im Zusammenspiel mit der Prozedur *GetDialogEnableStruct* (siehe später) ein bedingtes Aktiv- bzw. Passivsetzen von Dialogelementen (Beispiel: Ein

Fließkomma-Parameter soll nur modifizierbar sein, wenn ein bestimmter Schalter im Dialog gesetzt ist). Dazu enthält *TDialogEnableStruct* für jedes der jeweils 32 Eingabefelder eines Variablentyps ein entsprechendes Bit, das gesetzt (Eingabefeld wird aktiv) bzw. gelöscht (Eingabefeld wird passiv) wird. Das Feld *AllowD* legt schließlich fest, ob der Name der optionalen Parameterdatei (Variable *D* aus *TParameterStruct*) aktiv ist.

Nachfolgende Grafik zeigt beispielhaft die Palettenseite *Fließkommaparameter* des Parameterdialogs eines User-DLL-Blocks zur Reibungssimulation, der sich unter dem Namen REIBUNG.DLL im Verzeichnis *UserDLLs* (Quelltext REIBUNG.DPR im Unterverzeichnis *Sources*) befindet. Hier wurden u. a. vier Fließkomma-Parameter definiert, nur der zweite und dritte sind z. Z. aktiv.



Beispiel für den Parameterdialog eines User-Blocks mit vier Fließkomma-Parametern

Die letzten erforderlichen Datenstrukturen betreffen die Definition der Ein- und/oder Ausgänge des User-Blocks. Zunächst ist die Struktur *TNumberOfInputsOutputs* zu deklarieren. In dieser Struktur sind die Anzahl der Ein- bzw. Ausgänge des Blocks sowie ihre Namen (jeweils max. 40 Zeichen) festgelegt:

```
type
  PNumberOfInputsOutputs = ^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs = packed record
    Inputs  :Byte;                          {Anzahl Eingänge}
    Outputs :Byte;                          {Anzahl Ausgänge}
    NameI : Array[0..49] of String[40];     {Namen der Eingänge}
    NameO : Array[0..49] of String[40];     {Namen der Ausgänge}
  end;
```

Struktur von TNumberOfInputsOutputs *(in Pascal)*

Schließlich bleibt die Deklaration der Datentypen für die spätere Übergabe der aktuellen Blockein- und -ausgangswerte. Dazu dienen die Datentypen *TInputArray* bzw. *TOutputArray*:

```
type
  PInputArray = ^TInputArray;
  TInputArray = packed Array[1..50] of extended;   {Eingangswerte}
  POutputArray = ^TOutputArray;
  TOutputArray = packed Array[1..50] of extended;  {Ausgangswerte}
```

Datentypen TInputArray *bzw.* TOutputArray *(in Pascal)*

Damit sind alle erforderlichen Typendeklarationen erläutert. Die Deklaration in C erfolgt analog; nachfolgendes Listing faßt alle Datentypen dafür zusammen.

```
typedef struct{
    char NuE;
    char NuI;
    char NuB;
    long double E[32];
    int  I[32];
    char B[32];
    char D[256];
    long double EMin[32];
    long double EMax[32];
    int  IMin[32];
    int  IMax[32];
    char NaE[32][41];
    char NaI[32][41];
    char NaB[32][41];
    void FAR *UserDataPtr;
    void FAR *ParentPtr;
    unsigned int ParentHWnd;
    char *ParentName;
    unsigned int UserHWindow;
    FILE *DataFile
  } TParameterStruct, FAR *PParameterStruct;

typedef struct {
    long AllowE;
    long AllowI;
    long AllowB;
    char AllowD;
  } TDialogEnableStruct, FAR *PDialogEnableStruct;

typedef struct {
    char Inputs;
    char Outputs;
    char NameI[50][41];
    char NameO[50][41];
  } TNumberOfInputsOutputs, FAR *PNumberOfInputsOutputs;
```

```
typedef long double TInputArray[50];
typedef long double TOutputArray[50];
```

Datenschnittstelle des User-Blocks in der Sprache C

Die Funktionsschnittstelle des User-Blocks

Die Funktionsschnittstelle der User-DLL enthält die für den User-Block zu definierenden Funktionen bzw. Prozeduren. Diese unterscheiden sich in solche Funktionen, die zur Verwaltung des User-Blocks bzw. seines Parameterdialogs erforderlich sind (*organisierende* Funktionen) und die Funktionen, die die eigentliche Funktionalität des Blocks beschreiben, d. h. während der Simulation bzw. unmittelbar davor oder danach aufgerufen werden (*simulierende* Funktionen). Folgende Auflistung gibt zunächst eine Übersicht über alle erlaubten Funktionen mit ihren Parametern, bevor dann die Bedeutung aller Funktionen im Detail erläutert wird. Alle Funktionen sind als *stdcall* zu deklarieren.

Organisierende Funktionen:

```
Procedure IsUserDLL32;                                          (*)

Procedure GetParameterStruct(D:PParameterStruct);               (*)

Procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct);           (*)

Procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);   (*)
Procedure GetNumberOfInputsOutputs2(D:PNumberOfInputsOutputs;
                                    ParameterFileName: PChar;
                                    UserDataPtr: Pointer;
                                    var UserHWindow: THandle);

Procedure InitUserDLL(D:PParameterStruct);
Procedure DisposeUserDLL(D:PParameterStruct);

Procedure InitUserData(D: PParameterStruct);
Procedure DisposeUserData(D: PParameterStruct);

Procedure DialogOK(D:PParameterStruct);

Procedure ShowWindowDLL(D: PParameterStruct);
Procedure HideWindowDLL(D: PParameterStruct);

Function  SetInputChar: PChar;
Function  SetOutputChar: PChar;

Function  GetDLLName: PChar;
```



```
Procedure WriteToFile(AFileHandle: word; D: PParameterStruct);
Procedure ReadFromFile(AFileHandle: word; D: PParameterStruct);
Function  NumberOfLinesInSystemFile: word;

Procedure CallParameterDialogDLL(D1: PParameterStruct;
                                 D2: PNumberOfInputsOutputs);
```

Simulierende Funktionen:

```
Function  CanSimulateDLL(D:PParameterStruct):Integer;

Procedure InitSimulationDLL(D:PParameterStruct;
                            Inputs:PInputArray;
                            Outputs:POutputArray);

Procedure SimulateDLL(T:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray);
Procedure SimulateDLL2(T, DeltaT:Extended; D:PParameterStruct;
                       Inputs:PInputArray;
                       Outputs:POutputArray);

Procedure EndSimulationDLL;
Procedure EndSimulationDLL2(D: PParameterStruct);

Procedure SetEnhancedInformation(DeltaT, TSimu: extended;
                                 D: PParameterStruct);
```

Hinweis: Die mit einem Stern (*) gekennzeichneten Funktionen müssen in jeder User-DLL *auf jeden Fall* exportiert werden! Sofern Sie diese Funktionen nicht benötigen, lassen Sie die entsprechenden Funktionskerne einfach leer.

Bei der nachfolgenden Beschreibung der einzelnen Funktionen ist jeweils sowohl die Pascal-Signatur (oben) als auch die C-Signatur angegeben.

IsUserDLL32

Signatur

```
procedure IsUserDLL32; export stdcall;

void IsUserDLL32
```

Funktion Diese Dummy-Funktion wird lediglich exportiert, um die DLL als BORIS-32-Bit-User-DLL zu kennzeichnen. Der eigentliche Funktionsrumpf kann leer bleiben.

GetParameterStruct

Signatur

```
procedure GetParameterStruct(D:PParameterStruct);
                                    export stdcall;
```

```
void GetParameterStruct(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion In *GetParameterStruct* legen Sie die Blockparameter sowie die Daten zum Initialisieren des Parameterdialogs fest. Es werden Namen für Parameter vergeben, die Parameter selber werden initialisiert, obere und untere Eingabegrenzen eines Parameters festgelegt und letztlich wird die Anzahl der Parameter angegeben.

Beispiel:

Nachfolgendes Beispiel definiert einen User-Block mit vier Fließkomma-Parametern mit den Namen *Haftreibung*, *Gleitreibung*, *Normalkraft*, *Streuungsfaktor* sowie zwei Schalter-Parametern mit den Namen *Reale Reibung* bzw. *Stick and Slip aus Datei*. Außerdem wird eine zusätzliche Parameterdatei mit der Extension *SIM* zugelassen.

```
procedure GetParameterStruct(D:PParameterStruct); export stdcall;
var i : Integer;
begin
  D^.NuE:=4;  {Vier Fließkomma-Parameter}
  D^.NuI:=0;  {Keine Integer-Parameter}
  D^.NuB:=2;  {Zwei Schalter-Parameter}
  StrPCopy(D^.D,'*.sim');  {Dateien mit Endung SIM zulassen}
  {Initialisierung der verwendeten Daten}
  for i:=0 to 1 do begin
     D^.E[i]:=0;    {Fließkomma-Parameter 0 und 1 zu 0 initialisieren}
     D^.EMin[i]:=0; {Untere Grenze für Fließkomma-Par. 0 und 1 auf 0}
     D^.EMax[i]:=1; {Obere Grenze für Fließkomma-Par. 0 und 1 auf 1}
     D^.B[i]:=0;    {Schalter-Parameter zu 0 initialisieren}
  end;
  D^.E[2]:=0;          {Fließkomma-Parameter 2 zu 0 initialisieren}
  D^.EMin[2]:=0;       {Untere Grenze von Fließkomma-Par. 2 auf 0}
  D^.EMax[2]:=100000;  {Obere Grenze von Fließkomma-Par. 2 auf 100000}
  D^.E[3]:=0;          {Fließkomma-Parameter 3 zu 0 initialisieren}
  D^.EMin[3]:=0;       {Untere Grenze von Fließkomma-Par. 3 auf 0}
  D^.EMax[3]:=1;       {Untere Grenze von Fließkomma-Par. 3 auf 1}
  {Namensgebung max. 40 Zeichen !}
  strPCopy(D^.NaE[0],'Haftreibung:');     {Name für Fließkomma-Par. 0}
  strPCopy(D^.NaE[1],'Gleitreibung:');    {Name für Fließkomma-Par. 1}
  strPCopy(D^.NaE[2],'Normalkraft:');     {Name für Fließkomma-Par. 2}
  strPCopy(D^.NaE[3],'Streuungsfaktor:'); {Name für Fließkomma-Par. 3}
  strPCopy(D^.NaB[0],'Reale Reibung');    {Name für Schalter-Par. 0}
  strPCopy(D^.NaB[1],'"Stick and Slip" aus Datei'); {N. f. Sch.-P. 1}
end;
```

Beispiel für die Verwendung der Funktion GetParameterStruct

GetDialogEnableStruct

Signatur

```
procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct);
                                export stdcall;

void GetDialogEnableStruct
                          (PDialogEnableStruct D,
                           PParameterStruct D2)
```

Parameter *D* ist ein Zeiger auf die Struktur *TDialogEnableStruct.*

D2 ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion *GetDialogEnableStruct* legt die zugänglichen bzw. gesperrten (grau dargestellten) Elemente des Parametrierungsdialogs fest, d. h. welche Dialogelemente vom Anwender in welcher Situation angewählt werden können und welche nicht. Die Funktion wird bei Aufruf des Dialogs sowie *nach jeder Eingabe* aufgerufen, die vom Anwender im Dialog vorgenommen wird. Somit kann sie ggf. unmittelbar auf diese reagieren.

Beispiel:

Bei einem User-Block mit Fließkomma- und Schalter-Parametern sollen alle Fließkomma- und Schalter-Parameter jederzeit anwählbar sein. Der Dateiname für die optionale Parameterdatei soll aber nur zugänglich sein, wenn beide Schalter-Parameter gesetzt sind.

```
procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct); export stdcall;
begin
  D^.AllowE:=$FFFFFFFF;  {alle Fließkomma-Parameter zugänglich}
  D^.AllowB:=$FFFFFFFF   {alle Schalter-Parameter zugänglich};
  {Dateiname nur zugänglich, wenn beide Schalter-Parameter
  gesetzt sind!}
  D^.AllowD:= Byte(D2^.B[0] and D2^.B[1]);
end;
```

Beispiel für die Verwendung der Funktion GetDialogEnableStruct

Natürlich ist es auch möglich, z. B. nur ein Dialogelement vom gesperrten in den zugänglichen Zustand (oder umgekehrt) zu überführen und die anderen unbeeinflußt zu lassen. Soll z. B. der dritte Fließkomma-Parameter (also Fließkomma-Parameter 2) aktiviert werden, alle anderen Fließkomma-Parameter aber ihren aktuellen Zustand beibehalten, so lautet die entsprechende Anweisung

```
...
AllowE := AllowE or $00000004;   {Fließkomma-Parameter 2 freigeben}
...
```

GetNumberOfInputsOutputs

Signatur

```
procedure GetNumberOfInputsOutputs
          (D:PNumberOfInputsOutputs); export stdcall;
```

```
void GetNumberOfInputsOutputs(PNumberOfInputsOutputs D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TNumberOfInputsOutputs.*

Funktion In *GetNumberOfInputsOutputs* werden Anzahl und Namen der Ein- und Ausgänge des Blocks festgelegt. Die Namen werden später in den Listboxen *Eingangsvariable(n)* und *Ausgangsvariable(n)* des User-DLL-Dialogs angezeigt.

Beispiel:

Folgendes Listing zeigt die Realisierung der Funktion für einen User-Block mit zwei Eingängen und einem Ausgang.

```
procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
                                                        export stdcall;
begin
  D^.Inputs:=2;   { Der Block soll zwei Eingänge und}
  D^.Outputs:=1;  {           einen Ausgang haben!}
  {Namensgebung der Ein- und Ausgänge}
  StrPCopy(D^.NameI[0],'Sollwert');    {Name für ersten Eingang}
  StrPCopy(D^.NameI[1],'Istwert');     {Name für zweiten Eingang}
  StrPCopy(D^.NameO[0],'Stellgröße');  {Name für Ausgang}
end;
```

Beispiel für die Verwendung der Funktion GetNumberOfInputsOutputs

GetNumberOfInputsOutputs2

Signatur

```
Procedure GetNumberOfInputsOutputs2
          (D:PNumberOfInputsOutputs;
           ParameterFileName: PChar;
           UserDataPtr: Pointer;
           var UserHWindow: THandle); export stdcall
```

```
void GetNumberOfInputsOutputs2
                (PNumberOfInputsOutputs D,
                 char* ParameterFileName,
                 void* UserDataPtr,
                 int* UserHWindow)
```


Parameter *D* ist ein Zeiger auf die Struktur *TNumberOfInputsOutputs.*

ParameterFileName ist ein Zeiger auf das Feld *D* von *TParameterStruct.*

UserDataPtr ist ein Zeiger auf das Feld *UserDataPtr* von *TParameterStruct.*

UserHWindow ist ein Zeiger auf das Feld *UserHWindow* von *TParameterStruct.*

Funktion Diese Funktion kann alternativ zu *GetNumberOfInputsOutputs* benutzt werden, wenn die Anzahl der Ein- und Ausgänge variabel sein soll.

Beispiel:

Folgendes Listing zeigt die Realisierung der Funktion für einen User-Block, bei dem die Anzahl der Ein- und Ausgänge aus der Datei *ParameterFileName* gelesen werden sollen.

```
Procedure GetNumberOfInputsOutputs2(D:PNumberOfInputsOutputs;
           ParameterFileName: PChar; UserDataPtr: Pointer;
           var UserHWindow: THandle); export stdcall
var ParFile: TextFile;
    nIn, nOut, i: integer
begin
  AssignFile(ParFile, ParameterFileName);  //Dateinamen zuweisen
  ResetFile(ParFile);                      //Datei öffnen
  readln(ParFile, nIn, nOut);              //Anzahl Ein-/Ausgänge einlesen
  D^.Inputs  := nIn;                       //...und dem Block zuweisen
  D^.Outputs := nOut;
  {Namensgebung der Ein- und Ausgänge}
  for i:=1 to nIn  do StrPCopy(D^.NameI[i-1], 'Eingang' + IntToStr(i));
  for i:=1 to nOut do StrPCopy(D^.NameO[i-1], 'Ausgang' + IntToStr(i));
end;
```

Beispiel für die Verwendung der Funktion GetNumberOfInputsOutputs2

InitUserDLL
DisposeUserDLL

Signatur

```
procedure InitUserDLL(D:PParameterStruct);
                                        export stdcall;
procedure DisposeUserDLL(D:PParameterStruct);
                                        export stdcall;

void InitUserDLL(PParameterStruct D)
void DisposeUserDLL(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion *InitUserDLL* wird von BORIS unmittelbar nach der Initialisierung eines User-Blocks aufgerufen. Sie können diese Funktion beispielsweise nutzen, um dynamischen Speicher für Daten anzulegen, die Sie nicht in der Parameterstruktur *TParameterStruct* des Blocks unterbringen wollen oder können. Dies werden in der Regel z. B. Zustandsgrößen, Zwischenwerte irgendwelcher Art oder umfangreichere Parameterstrukturen sein, die aus einer Parameterdatei gelesen wurden. Den Zeiger auf den hier angelegten dynamischen Speicher können Sie im Parameter *UserDataPtr* von *TParameterStruct* ablegen.

Ein anderer Anwendungsfall für diese Funktion sind User-Blöcke mit Visualisierungsfenster. Bei solchen Anwendungen kann das Visualisierungsfenster des User-Blocks in dieser Funktion generiert werden (siehe dazu das Beispiel *Füllstandsregelung* an späterer Stelle!).

Das Gegenstück zu *InitUserDLL* stellt *DisposeUserDLL* dar. Diese Funktion wird unmittelbar vor der Freigabe eines User-Blocks aufgerufen. Der in *InitUserDLL* angelegte Speicher muß daher in *DisposeUserDLL* wieder freigegeben werden.

Werden die erforderlichen Daten nicht während der gesamten "Lebensdauer" des User-Blocks, sondern nur während der eigentlichen Simulation selbst benötigt (dies wird in der Regel häufiger der Fall sein), können statt der Funktionen *InitUserDLL* und *DisposeUserDLL* auch die Funktionen *InitUserData* und *DisposeUserData* (werden nachfolgend beschrieben) benutzt werden.

Die Verwendung von *InitUserDLL* und *DisposeUserDLL* ist in der Regel nur in komplexeren User-Blöcken erforderlich.

Beispiel:

Es soll ein User-Block erstellt werden, der neben den Blockparametern eine 20x20-Matrix vom Typ *Extended* sowie einen Vektor der Dimension 50 vom Typ *Integer* benötigt. Folgendes Listing zeigt die entsprechende Verwendung der Funktionen *InitUserDLL* und *DisposeUserDLL*.

```
...
...
type
```



```
  {Deklaration des Datentyps für die User-Daten}
  PUserData = ^TUserData;
  TUserData = record
    Matrix: array[1..20, 1..20] of extended;
    Vektor: array[1..50] of integer;
  end;


procedure InitUserDLL(D: PParameterStruct); export stdcall;
begin
  ...
  ...
  {Speicher für User-Daten anfordern}
  GetMem(D^.UserDataPtr, SizeOf(TUserData));
  ...
  ...
end {of InitUserDLL};


procedure DisposeUserDLL(D: PParameterStruct); export stdcall;
begin
  ...
  ...
  {Speicher für User-Daten wieder freigeben}
  FreeMem(D^.UserDataPtr, SizeOf(TUserData));
  ...
  ...
end {of DisposeUserDLL};
...
...
```

Beispiel für die Verwendung von InitUserDLL *und* DisposeUserDLL

Soll während der Simulation dann z. B. auf ein bestimmtes Matrixelement zugegriffen werden, lautet die entsprechende Pascal-Anweisung:

```
...
...
{2. Zeile, 3. Spalte der Matrix auf 5 setzen}
PUserData(UserDataPtr)^.Matrix[2, 3] := 5;
...
...
```

Zugriff auf User-Daten

InitUserData
DisposeUserData

Signatur

```
procedure InitUserData(D:PParameterStruct);
                                          export stdcall;
procedure DisposeUserData(D:PParameterStruct);
                                          export stdcall;

void InitUserData(PParameterStruct D)
void DisposeUserData(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion *InitUserData* und *DisposeUserData* entsprechen von ihrem Anwendungsbereich her den zuvor beschriebenen Funktionen *InitUserDLL* und *DisposeUserDLL*, werden aber nicht bei der Initialisierung bzw. Freigabe des User-Blocks, sondern unmittelbar vor bzw. nach der Simulation (d. h. direkt vor *InitSimulationDLL* bzw. nach *EndSimulationDLL*) aufgerufen. Sie eignen sich damit besonders für die Verwaltung von Daten, die nur während der Simulation selbst benötigt werden (siehe dazu das Beispiel *Kennfeld-DLL* an späterer Stelle). Auch diese Funktionen sind für die Realisierung einfacher DLLs nicht erforderlich.

DialogOK

Signatur

```
procedure DialogOK(D:PParameterStruct);
                                          export stdcall;

void DialogOK(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion *DialogOK* ist in der Regel nur bei User-Blöcken mit Visualisierungsfenster erforderlich und dort auch nur in wenigen Fällen. Diese Funktion wird von BORIS aufgerufen, wenn der Anwender den Parameterdialog des User-Blocks über den *OK*-Schalter verlassen hat. Sie kann beispielsweise benutzt werden, um den Inhalt des Visualisierungsfensters an die geänderten Blockparameter anzupassen.

Beispiel:

Folgendes Listing zeigt ein Beispiel für die Anwendung von *DialogOK*.

```
procedure DialogOK(D:PParameterStruct); export stdcall;
begin
  {Nachdem der Parameterdialog über OK verlassen wurde,
  soll das Visualisierungsfenster, dessen Handle in
  UserHWindow liegt, aufgefrischt werden!}
  InvalidateRect(D^.UserHWindow, nil, true);
end;
```

Beispiel für die Anwendung von DialogOK

ShowWindowDLL
HideWindowDLL

Signatur

```
Procedure ShowWindowDLL(D: PParameterStruct);
                                    export stdcall;
Procedure HideWindowDLL(D: PParameterStruct);
                                    export stdcall;

void ShowWindowDLL(PParameterStruct D)
void HideWindowDLL(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion Diese Funktionen werden nur bei User-DLLs mit Visualisierungsfenstern benötigt. Sie können benutzt werden, um das Visualisierungsfenster anzuzeigen bzw. zum Symbol zu verkleinern. Die Funktionen werden von BORIS aufgerufen, wenn der Anwender die Option OPTIONEN | ALLE ANZEIGEFENSTER ZEIGEN bzw. OPTIONEN | ALLE ANZEIGEFENSTER VERBERGEN anwählt.

Beispiel:

Folgendes Listing zeigt ein Anwendungsbeispiel für die beiden Funktionen:

```
...
procedure ShowWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Anzeigefenster in Normalgröße anzeigen}
  ShowWindow(D^.UserHWindow, sw_Normal);
end;

procedure HideWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Anzeigefenster zum Symbol verkleinern}
  ShowWindow(D^.UserHWindow, sw_Minimize);
end;
```

Beispiel für die Anwendung von ShowWindowDLL *bzw.* HideWindowDLL

SetInputChar
SetOutputChar

Signatur

```
Function SetInputChar: PChar; export stdcall;
Function SetOutputChar: PChar; export stdcall;

char SetInputChar(void)
char SetOutputChar(void)
```

Rückgabewert Der Rückgabewert enthält einen String, dessen Zeichen die Beschriftung der einzelnen Blockeingänge bzw. Blockausgänge festlegen.

Funktion *SetInputChar* bzw. *SetOutputChar* können benutzt werden, um die standardmäßige Beschriftung der Blockeingänge bzw. -ausgänge des User-Blocks in der Strukturdarstellung zu ändern. In der Regel wird der Eingang eines Blocks mit *E* (bei nur einem Eingang) und der Ausgang mit *A* (bei nur einem Ausgang) beschriftet, bei mehreren Ein- bzw. Ausgängen werden diese durchnumeriert. Durch Implementierung von *SetInputChar* bzw. *SetOutputChar* kann für jeden Ein- bzw. Ausgang stattdessen ein benutzerdefiniertes Zeichen vorgegeben werden.

Beispiel:

Bei einem User-Block mit zwei Eingängen und einem Ausgang sollen die Eingänge mit *w* und *x* und der Ausgang mit *y* beschriftet werden. Nachfolgendes Listing zeigt die entsprechende Implementierung der Funktionen *SetInputChar* und *SetOutputChar*.

```
Function SetInputChar: PChar; export stdcall;
begin
  SetInputChar := 'wx'; {Eingang 1 mit "w", Eingang 2 mit "x" beschriften}
end;

Function SetOutputChar: PChar; export stdcall;
begin
  SetOutputChar := 'y'; {Ausgang mit "y" beschriften}
end;
```

Beispiel für die Anwendung von SetInputChar *und* SetOutputChar

GetDLLName

Signatur

```
Function GetDLLName: PChar; export stdcall;
char GetDLLName(void)
```

Rückgabewert Bezeichnung des User-DLL-Blocks

Funktion Über diese Funktion kann dem User-DLL-Block ein Name gegeben werden, der dann im Untermenü SYSTEMBLÖCKE | USER-DLL-BLÖCKE bzw. im ToolTip-Fenster der Palettenseite *User* der

Systemblock-Toolbar erscheint.

Beispiel:

Nachfolgend definierter User-DLL-Block erhält die Bezeichnung *Riesen-Display*.

```
function GetDLLName: PChar; export stdcall;
begin
  GetDLLName := 'Riesen-Display';
end;
```

Beispiel für die Realisierung von GetDLLName

CallParameterDialogDLL

Signatur

```
Procedure CallParameterDialogDLL(D1: PParameterStruct;
            D2: PNumberOfInputsOutputs);export stdcall;

void CallParameterDialogDLL
     (PParameterStruct D1, PNumberOfInputsOutputs D2)
```

Parameter *D1* ist ein Zeiger auf die Struktur *TParameterStruct.*

D2 ist ein Zeiger auf die Struktur *TNumberOfInputsOutputs.*

Funktion Diese Funktion ermöglicht den Aufruf eines benutzerdefiniertes Parameterdialogs anstelle des Standard-Dialogs.

Beispiel:

Nachfolgendes Listing aus der Beispiel-DLL DLLDEMO2 zeigt den Aufruf eines benutzerdefinierten Dialogs in DELPHI.

```
procedure CallParameterDialogDLL(D1:PParameterStruct;
                                 D2:PNumberOfInputsOutputs);export stdcall;
(* Stellt den benutzerdefinierten Parameterdialog dar *)
var s: string;
    Code: integer;
begin
  Form1 := TForm1.Create(Application);
  Str(D1^.E[0]:10, s);
  Form1.Edit1.Text := s;
  Form1.Checkbox1.Checked := D1^.B[0] > 0;
  if Form1.ShowModal = mrOk then begin
    Val(Form1.Edit1.Text, D1^.E[0], Code);
    D1^.B[0] := ord(Form1.Checkbox1.Checked);
  end;
  Form1.Free;
end;
```

Beispiel für die Realisierung von CallParameterDialogDLL

WriteToFile
ReadFromFile
NumberOfLinesInSystemFile

Signatur

```
Procedure WriteToFile(AFileHandle: word;
            D: PParameterStruct); export stdcall
Procedure ReadFromFile(AFileHandle: word;
            D: PParameterStruct); export stdcall
Function  NumberOfLinesInSystemFile: word;
                                    export stdcall

void WriteToFile(int AFileHandle,
                              PParameterStruct D)
void ReadFromFile(int AFileHandle,
                              PParameterStruct D)

int NumberOfLinesInSystemFile(void);
```

Parameter *AFileHandle* ist das Handle der Systemdatei.

D ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion Diese drei Funktionen dienen zur Speicherung von blockspezifischen Parametern, die über die 32 Fließkomma-, Ganzzahl- und Schalterparameter hinausgehen. Sie sind daher nur in seltenen Fällen notwendig. Die User-DLL BIGDIS32.DLL im Verzeichnis *UserDLLs* demonstriert die Anwendung der drei Funktionen.

CanSimulateDLL

Signatur

```
function CanSimulateDLL(D:PParameterStruct):Integer;
                                        export stdcall;

int CanSimulateDLL(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Rückgabewert Der Rückgabewert '0' zeigt an, daß nicht simuliert werden kann, der Rückgabewert von '1', daß dieser Block simulationsbereit ist.

Funktion *CanSimulateDLL* wird von BORIS vor Beginn der Simulation (noch vor *InitSimulationDLL*) aufgerufen, um zu überprüfen, ob der User-Block simuliert werden kann. Mittels dieser Funktion ist es also möglich, unter bestimmten Bedingungen (beispielsweise, wenn der Block eine Parameterdatei benötigt und für diese ein nicht existierender Dateiname angegeben wurde) eine Simulation zu verhindern, indem der Rückgabewert auf 0 gesetzt wird. In den meisten Fällen sind derartige Überprüfungen nicht notwendig und

der Rückgabewert kann auf 1 gesetzt werden.

Beispiel:

Folgende Listings zeigen zwei Realisierungsmöglichkeiten für die Funktion *CanSimulateDLL*.

```
Function CanSimulateDLL(D: PParameterStruct): integer; export stdcall;
begin
  CanSimulateDLL := 1; {Simulation immer zulässig!}
end;
```

```
Function CanSimulateDLL(D: PParameterStruct): integer; export stdcall;
begin
  {Die Simulation soll nur zulässig sein, wenn Fließkomma-Parameter 0
  und Fließkomma-Parameter 1 unterschiedliches Vorzeichen haben!}
  if D^.E[0] * D^.E[1] < 0 then
    CanSimulateDLL := 1
  else
    CanSimulateDLL := 0;
end;
```

Beispiele für die Realisierung von CanSimulateDLL

InitSimulationDLL

Signatur

```
procedure InitSimulationDLL (D:PParameterStruct;
                             Inputs:PInputArray;
                             Outputs:POutputArray);
                             export stdcall;

void InitSimulationDLL (PParameterStruct D,
                        TInputArray FAR Inputs,
                        TOutputArray FAR Outputs)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Inputs ist ein Zeiger auf die Struktur *TInputArray.*

Outputs ist ein Zeiger auf die Struktur *TOutputArray.*

Funktion *InitSimulationDLL* initialisiert den Block für den Zeitpunkt $t = 0$ bzw. den ersten Simulationsschritt. Hier können also Voreinstellungen vorgenommen werden, die vor Beginn der eigentlichen Simulationsschleife notwendig werden (z. B. Setzen von Anfangswerten oder inneren Zustandsgrößen, Vorbelegung der Ausgangswerte usw.). Sofern erforderlich, können innerhalb dieser Funktion z. B. auch Daten, die für die Simulation benötigt werden, aus einer externen Parameterdatei gelesen werden (siehe Beispiel *Kennfeld-DLL* an späterer Stelle). Die aktuellen Eingangswerte

des Blocks werden über den Zeiger *Inputs* übergeben; die berechneten Ausgangswerte müssen über den Zeiger *Outputs* zurückgeliefert werden.

Beispiel:

Nachfolgendes Listing zeigt eine beispielhafte Realisierung von *InitSimulationDLL* für einen Block mit zwei Eingängen und zwei Ausgängen. In diesem Fall soll zum Zeitpunkt $t = 0$ der erste Eingang auf den ersten Ausgang durchgeschaltet werden und der zweite Ausgang zu 0 initialisiert werden.

```
procedure InitSimulationDLL (D:PParameterStruct; Inputs:PInputArray;
                             Outputs:POutputArray); export stdcall;
begin
  Outputs^[1] := Inputs^[1];  {Ausgang 1 auf Eingang 1}
  Outputs^[2] := 0.0;         {Ausgang 2 zu 0 initialisieren}
end;
```

Beispiel für die Realisierung von InitSimulationDLL

SimulateDLL

Signatur

```
Procedure SimulateDLL(T:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray);
                      export stdcall;

void SimulateDLL(long double T,
                       PParameterStruct D,
                       TInputArray FAR Inputs,
                       TOutputArray FAR Outputs)
```

Parameter *T* ist der Zeitwert des augenblicklichen Simulationsschrittes

D ist ein Zeiger auf die Struktur *TParameterStruct.*

Inputs ist ein Zeiger auf die Struktur *TInputArray.*

Outputs ist ein Zeiger auf die Struktur *TOutputArray*.

Funktion *SimulateDLL* enthält den eigentlichen numerischen Kern des User-Blocks, d. h. den zu jedem Simulationsschritt abzuarbeitenden Algorithmus. Diese Funktion wird zu jedem Simulationsschritt aufgerufen. Der Parameter *T* enthält dabei den momentanen Zeitwert, der Parameter *D* einen Zeiger auf die Blockparameter. Die aktuellen Eingangswerte des Blocks werden über den Zeiger *Inputs* übergeben; die berechneten Ausgangswerte müssen über den Zeiger *Outputs* zurückgeliefert werden.

Wird im Simulationsalgorithmus auch die Simulationsschrittweite ΔT benötigt, kann statt der Funktion *SimulateDLL* die Funktion *SimulateDLL2* (wird nachfolgend beschrieben) benutzt werden, der die Schrittweite als zusätzlicher Parameter übergeben wird.

Beispiel:

Folgendes Listing zeigt die Realisierung von *SimulateDLL* für einen Block mit zwei Eingängen und zwei Ausgängen. Am ersten Ausgang wird der arithmetische Mittelwert beider Eingänge - zusätzlich multipliziert mit Fließkomma-Parameter 0 - und am zweiten Ausgang der geometrische Mittelwert - multipliziert mit Fließkomma-Parameter 1 - ausgegeben.

```
Procedure SimulateDLL(T:Extended; D:PParameterStruct;
                      Inputs:PInputArray;
                      Outputs:POutputArray); export stdcall;
begin
  Outputs^[1] := D^.E[0] * (Inputs^[1] + Inputs^[2]) | 2;
  Outputs^[2] := D^.E[1] * sqrt((Inputs^[1] * Inputs^[2]));
end;
```

Beispiel für die Realisierung von SimulateDLL

EndSimulationDLL

Signatur

```
procedure EndSimulationDLL; export stdcall;
```

```
void EndSimulationDLL(void)
```

Funktion *EndSimulationDLL* wird aufgerufen, sobald die Simulation beendet wurde. In dieser Funktion können somit z. B. geöffnete Dateien geschlossen werden. In den meisten Anwendungen kann diese Funktion jedoch leer bleiben.

Statt *EndSimulationDLL* kann auch die Funktion *EndSimulationDLL2* (wird nachfolgend beschrieben) benutzt werden, der als zusätzlicher Parameter ein Zeiger auf die Struktur *TParameterStruct* übergeben wird.

Beispiel:

Nachfolgendes Listing zeigt ein Beispiel für die Anwendung von *EndSimulationDLL*.

```
procedure EndSimulationDLL; export stdcall;
begin
  Close(DataFile);   {Parameterdatei schließen}
end;
```

Beispiel für die Anwendung von EndSimulationDLL

SimulateDLL2

Signatur

```
Procedure SimulateDLL2(T, DeltaT:Extended;
                       D:PParameterStruct;
                       Inputs:PInputArray;
                       Outputs:POutputArray);
                       export stdcall;

void SimulateDLL2(long double T,
                        long double DeltaT,
                        PParameterStruct D,
                        TInputArray FAR Inputs,
                        TOutputArray FAR Outputs)
```

Parameter *T* ist der Zeitwert des augenblicklichen Simulationsschrittes

DeltaT ist die Simulationsschrittweite.

D ist ein Zeiger auf die Struktur *TParameterStruct.*

Inputs ist ein Zeiger auf die Struktur *TInputArray.*

Outputs ist ein Zeiger auf die Struktur *TOutputArray.*

Funktion *SimulateDLL2* kann statt der Funktion *SimulateDLL* benutzt werden, wenn zur Verarbeitung auch die Simulationsschrittweite ΔT benötigt wird.

EndSimulationDLL2

Signatur

```
procedure EndSimulationDLL2(D: PParameterStruct);
                                    export stdcall;

void EndSimulationDLL2(PParameterStruct D)
```

Parameter *D* ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion *EndSimulationDLL2* kann statt der Funktion *EndSimulationDLL* benutzt werden, wenn zur Verarbeitung auch die Parameterstruktur *TParameterStruct* benötigt wird.

SetEnhancedInformation

Signatur

```
procedure SetEnhancedInformation
                                  (DeltaT, TSimu: extended;
                                   D: PParameterStruct);
                                   export stdcall;

void SetEnhancedInformation
                                    (long double DeltaT,
                                     long double TSimu,
                                     PParameterStruct D)
```

Parameter *DeltaT* ist die Simulationsschrittweite.

TSimu ist die Simulationsdauer.

D ist ein Zeiger auf die Struktur *TParameterStruct.*

Funktion *SetEnhancedInformation* wird nur in seltenen Fällen benötigt. Die Funktion liefert dem User-Block die eingestellte Simulationsschrittweite und -dauer zur weiteren Verwendung. Beide Größen können z. B. in nicht benutzten Fließkomma-Parametern zur weiteren Nutzung abgelegt werden.

Beispiel:

In nachfolgendem Beispiel werden *DeltaT* und *TSimu* in den Fließkomma-Parametern 3 und 4 gespeichert.

```
procedure SetEnhancedInformation(DeltaT, TSimu: extended;
                                 D: PParameterStruct); export stdcall;
begin
  D^.E[3] := DeltaT;
  D^.E[4] := TSimu;
end;
```

Beispiel für die Anwendung von SetEnhancedInformation

Beispiele

Beispiel 1: Eine simple User-DLL

Wir wollen zunächst beginnen mit einem einfachen Beispiel einer User-DLL mit zwei Eingängen und einem Ausgang. Der Block soll in Abhängigkeit vom

ersten Eingang (Steuereingang) x_1 den Wert des zweiten Eingangs (Dateneingang) x_2 mit unterschiedlichen Verstärkungenfaktoren k_1, k_2, k_3 multiplizieren und am Ausgang *y* ausgeben. Über einen Schalter-Parameter b_1 soll der Ausgang zusätzlich invertiert werden können:

- Falls x_1 negativ ist, soll x_2 mit k_1 multipliziert werden.
- Falls x_1 null ist, soll x_2 mit k_2 multipliziert werden.
- Falls x_1 positiv ist, soll x_2 mit k_3 multipliziert werden.
- Falls b_1 gesetzt ist, soll der Ausgang zusätzlich invertiert werden.

Nachfolgendes Listing zeigt die Realisierung der DLL in Pascal. Die fertig compilierte DLL befindet sich als DLLDEMO1.DLL im *UserDLLs*-Verzeichnis, der zugehörige Quelltext unter DLLDEMO1.DPR im Unterverzeichnis *Sources*. Alle wesentlichen Kommentare zur DLL sind im Quelltext vermerkt.

```
Library DLLDemo1;
(***********************************************)
(*                                             *)
(*       Einfache Demo-DLL für BORIS           *)
(*       (Beispiel 1 des Handbuchs)            *)
(*                                             *)
(***********************************************)

uses WinProcs,
  SysUtils,
  WinTypes;

type
  PParameterStruct=^TParameterStruct;
  TParameterStruct=packed record
    NuE : Byte;                    {Anzahl reeller Zahlenwerte}
    NuI : Byte;                    {Anzahl ganzer Zahlenwerte}
    NuB : Byte;                    {Anzahl Schalter}
    E:Array[0..31] of Extended;    {reelle Zahlenwerte}
    I:Array[0..31] of Integer;     {ganze Zahlenwerte}
    B:Array[0..31] of Byte;        {Schalter}
    D:Array[0..255] of char;       {event. Dateiname für weitere Daten.}
    EMin:Array[0..31] of Extended; {untere Eingabegrenze}
    EMax:Array[0..31] of Extended; {obere Eingabegrenze}
    IMin:Array[0..31] of Integer;  {untere Eingabegrenze}
    IMax:Array[0..31] of Integer;  {obere Eingabegrenze}
    NaE : Array[0..31,0..40] of char;    {Namen der reellen Zahlenwerten}
    NaI : Array[0..31,0..40] of char;    {Namen der ganzen Zahlenwerten}
    NaB : Array[0..31,0..40] of char;    {Namen der Schalter}
    {den Rest der Variablen von TParameterStruct können wir uns sparen,
     da wir ihn nicht benötigen!}
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;               { Soll die Eingabe eines Wertes   }
```



```
    AllowI: Longint;                { un-/zulässig sein so ist das Bit}
    AllowB: Longint;                { des Allow?-Feldes 0 bzw. 1}
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                   {Anzahl Eingänge}
    Outputs:Byte;                   {Anzahl Ausgänge}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;

  PInputArray = ^TInputArray;
  TInputArray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputArray = packed array[1..50] of extended;


procedure GetParameterStruct(D:PParameterStruct);export stdcall;
begin
  D^.NuE:=3; {3 Fließkomma-Parameter}
  D^.NuI:=0; {0 Integer-Parameter}
  D^.NuB:=1; {1 Schalter-Parameter}
  {Namen der Parameter}
  StrCopy(D^.NaE[0], 'Verstärkung k1 für Steuereing. < 0');
  StrCopy(D^.NaE[1], 'Verstärkung k2 für Steuereing. = 0');
  StrCopy(D^.NaE[2], 'Verstärkung k3 für Steuereing. > 0');
  StrCopy(D^.NaB[0], 'Ausgang invertiert');
  {Parameter initialisieren}
  D^.E[0] := 1;
  D^.E[1] := 5;
  D^.E[2] := 10;
  D^.B[0] := 0;
  {Grenzwerte für Parameter festlegen}
  D^.EMin[0] := 0.0; D^.EMax[0] := 100000;
  D^.EMin[1] := 0.0; D^.EMax[1] := 100000;
  D^.EMin[2] := 0.0; D^.EMax[2] := 100000;
end;

procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct); export stdcall;
begin
  {Alle Dialogelemente sollen jederzeit zugänglich sein}
  D^.AllowE := $FFFFFFFF;
  D^.AllowI := $FFFFFFFF;
  D^.AllowB := $FFFFFFFF;
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
                                                    export stdcall;
begin
  D^.Inputs:=2;            { Der Block soll zwei Eingänge und}
  D^.Outputs:=1;           {            einen Ausgang haben !}
  StrPCopy(D^.NameI[0],'Dateneingang');    {Namensgebung des 1. Eingangs}
  StrPCopy(D^.NameI[1],'Steuereingang');   {Namensgebung des 2. Eingangs}
  StrPCopy(D^.NameO[0],'Ausgang');         {Namensgebung des 1. Ausgangs}
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
begin
  {Simulation immer zulässig!}
  CanSimulateDLL := 1;
end;
```

```
procedure SimulateDLL(T:Extended;D:PParameterStruct;Inputs:PInputArray;
                      Outputs:POutputArray);export stdcall;
begin
  if Inputs^[1] < 0 then {Eingang 1 negativ}
    Outputs^[1] := D^.E[0] * Inputs^[2]
  else if Inputs^[1] = 0 then {Eingang 1 null}
    Outputs^[1] := D^.E[1] * Inputs^[2]
  else {Eingang 1 positiv}
    Outputs^[1] := D^.E[2] * Inputs^[2];
  if D^.B[0] = 1 then {Schalter-Parameter 0 = 1 ==> Ausgang invertieren}
    Outputs^[1] := -Outputs^[1];
end;

procedure InitSimulationDLL(D:PParameterStruct;Inputs:PInputArray;
                            Outputs:POutputArray);export stdcall;
begin
  {Der Initialisierungsschritt soll genauso durchgeführt werden wie alle
   Simulationsschritte. Also rufen wir einfach SimulateDLL auf!}
  SimulateDLL(0, D, Inputs, Outputs);
end;

procedure EndSimulationDLL; export stdcall;
begin
  {wird nicht benötigt}
end;

function SetInputChar: PChar; export stdcall;
begin
  {Steuereingang mit 'S', Dateneingang mit 'D' beschriften}
  SetInputChar := 'SD';
end;

procedure IsUserDLL32; export stdcall;
begin
end;

{Exportieren der notwendigen Funktionen und Prozeduren }
exports
  GetParameterStruct,
  GetDialogEnableStruct,
  GetNumberOfInputsOutputs,
  CanSimulateDLL,
  InitSimulationDLL,
  SimulateDLL,
  EndSimulationDLL,
  SetInputChar,
  IsUserDLL32;
begin
  {Initialisierung der DLL (nicht notwendig)}
end.
```

Pascal-Listing zu Beispiel 1

Beispiel 2: Benutzerdefiniertes Kennfeld

Als Erweiterung zur benutzerdefinierten Kennlinie, die in der BORIS-Systemblockbibliothek ja standardmäßig verfügbar ist, soll eine User-DLL zur Reali-

sierung eines benutzerdefinierten *Kennfelds* der Form $z = f(x, y)$ erstellt werden. Die User-DLL soll folgende Spezifikationen erfüllen:

- Das Kennfeld soll in Matrixform als FWM-Datei (siehe Abschnitt *WinFACT-Dateitypen* in Kapitel *2 Grundlagen*) eingelesen werden. Der Name der Datei soll über den Parameterdialog vorgebbar sein. Der zugrundeliegende Abszissenbereich $[\, x_{\min,} x_{\max}\,]$ bzw. $[\, y_{\min,} y_{\max}\,]$ soll ebenfalls über den Parameterdialog des User-Blocks einstellbar sein. Die Matrix soll maximal die Dimension 20×20 aufweisen können.
- Zwischen den Stützpunkten soll auf Wunsch linear interpoliert werden können. Liegt ein Eingangswertepaar (x, y) außerhalb des von der Stützstellenmatrix erfaßten Bereichs, soll extrapoliert werden.
- Neben dem eigentlichen *z*-Ausgang soll ein zweiter Ausgang anzeigen, ob ein Eingangswertepaar (x, y) außerhalb des von der Stützstellenmatrix erfaßten Bereichs liegt, also extrapoliert wird. In diesem Fall soll dieser Ausgang High-Pegel, sonst Low-Pegel aufweisen. Die Werte für Low- und High-Pegel sollen ebenfalls über den Parameterdialog einstellbar sein.

Nachfolgendes Listing zeigt die Realisierung in Pascal. Die fertig compilierte DLL befindet sich als DRDFELD.DLL im *UserDLLs*-Verzeichnis, der zugehörige Quelltext unter DRDFELD.DPR im Unterverzeichnis *Sources*. Alle wesentlichen Kommentare zur DLL sind im Quelltext vermerkt.

```
Library DRDFeld;

(************************************************)
(*                                              *)
(*          Kennfeld-User-DLL für BORIS         *)
(*           (Beispiel 2 des Handbuchs)         *)
(*                                              *)
(************************************************)

uses Windows, SysUtils;

type
  PParameterStruct=^TParameterStruct;
  TParameterStruct= packed record
    NuE : Byte;                     {Anzahl reeller Zahlenwerte}
    NuI : Byte;                     {Anzahl ganzer Zahlenwerte}
    NuB : Byte;                     {Anzahl Schalter}
    E:Array[0..31] of Extended;     {reelle Zahlenwerte}
    I:Array[0..31] of Integer;      {ganze Zahlenwerte}
    B:Array[0..31] of Byte;         {Schalter}
    D:Array[0..255] of char;        {event. Dateiname für weitere Daten.}
    EMin:Array[0..31] of Extended;
    EMax:Array[0..31] of Extended;
    IMin:Array[0..31] of Integer;
    IMax:Array[0..31] of Integer;
```

```
    NaE : Array[0..31,0..40] of char;
    NaI : Array[0..31,0..40] of char;
    NaB : Array[0..31,0..40] of char;
    UserDataPtr: Pointer;
    {Den Rest von TParameterStruct können wir uns sparen, da wir ihn
     nicht benötigen!}
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;                { Soll die Eingabe eines Wertes   }
    AllowI: Longint;                { un-/zulässig sein so ist das Bit}
    AllowB: Longint;                { des Allow?-Feldes 0 bzw. 1}
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                          {Anzahl Eingänge}
    Outputs:Byte;                          {Anzahl Ausgänge}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;

  PInputArray = ^TInputArray;
  TInputArray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputArray = packed array[1..50] of extended;


  {TUserData enthält die Kennfelddaten}
  TSingleMatrix = Array[1..20, 1..20] of Single;
  PUserData = ^TUserData;
  TUserData = record
                z: TSingleMatrix; {Funktionswertmatrix}
                nx,ny: Word;      {Anzahl Spalten bzw. Zeilen der Matrix}
                dx,dy: Extended;  {x- bzw. y-Abstand zwischen zwei
                                   Stützpunkten}
              end;

Const
  {Zwei Konstanten-Deklarationen für später...}
  SingleMin=-3.4E38;
  SingleMax= 3.4E38;


procedure GetParameterStruct(D:PParameterStruct);export stdcall;
begin
  D^.NuE:=6;               {Sechs Fließpunktparameter}
  D^.NuI:=0;               {Keine Ganzzahlparameter }
  D^.NuB:=1;               {Ein Schalter}
  StrPCopy(D^.D,'*.fwm');  {Dateien mit Endung fwm zulassen}
  {Initialisierung der verwendeten Daten}
  D^.B[0]:=0;
  D^.E[0]:=-1; D^.EMin[0]:=SingleMin; D^.EMax[0]:=SingleMax;
  D^.E[1]:=1;  D^.EMin[1]:=SingleMin; D^.EMax[1]:=SingleMax;
  D^.E[2]:=-1; D^.EMin[2]:=SingleMin; D^.EMax[2]:=SingleMax;
  D^.E[3]:=1;  D^.EMin[3]:=SingleMin; D^.EMax[3]:=SingleMax;
  D^.E[4]:=0;  D^.EMin[4]:=SingleMin; D^.EMax[4]:=SingleMax;
  D^.E[5]:=5;  D^.EMin[5]:=SingleMin; D^.EMax[5]:=SingleMax;
  {Namensgebung max. 40 Zeichen !}
  StrPCopy(D^.NaE[0],'x - Achsenanfang');
  StrPCopy(D^.NaE[1],'x - Achsenende');
  StrPCopy(D^.NaE[2],'y - Achsenanfang');
  StrPCopy(D^.NaE[3],'y - Achsenende');
```



```
  StrPCopy(D^.NaE[4],'Low-Pegel für Ausgang 2');
  StrPCopy(D^.NaE[5],'High-Pegel für Ausgang 2');
  StrPCopy(D^.NaB[0],'lineare Interpolation');
end;

procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct); export stdcall;
begin
  {Alle Dialogelemente sollen jederzeit zugänglich sein!}
  D^.AllowE:=$FFFFFFFF;
  D^.AllowB:=$FFFFFFFF;
  D^.AllowI:=$FFFFFFFF;
  D^.AllowD:= 1;
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
                                                              export stdcall;
begin
  D^.Inputs:=2;            { Der Block soll zwei Eingänge und}
  D^.Outputs:=2;           {                zwei Ausgänge haben !}
  StrPCopy(D^.NameI[0],'Abszissen-Wert (x)');    {Name des 1. Eingangs}
  StrPCopy(D^.NameI[1],'Abszissen-Wert (y)');    {Name des 2. Eingangs}
  StrPCopy(D^.NameO[0],'Ordinate (z) ');         {Name des 1. Ausgangs}
  StrPCopy(D^.NameO[1],'Extrapolation ON/OFF');  {Name des 2. Ausgangs}
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
var Datei: Text;
    Dateiname : Array[0..255] of char;
begin
  {Simulation nur zulassen, falls die angegebene FWM-Datei auch existiert!}
  StrCopy(Dateiname, D^.D);
  if Pos('*',StrPas(Dateiname))=0 then begin
    Assign(Datei,Dateiname);
    {$I-} Reset(Datei); {$I+}
    if IOResult = 0 then begin
      CanSimulateDLL := 1;
      Close(Datei);
    end else CanSimulateDLL := 0;
  end else CanSimulateDLL:=0;
end;

procedure SimulateDLL(T:Extended;D:PParameterStruct;
                    Inputs:PInputArray;Outputs:POutputArray);
                                                            export stdcall;
var ix, iy: integer;
    x0, y0, xp, yp: Extended;
    Extrapolation, Interpolation: boolean;
begin
  with PUserData(D^.UserDataPtr)^ do begin
    Extrapolation := (Inputs^[1]<D^.E[0]) or (Inputs^[1]>D^.E[1]) or
                     (Inputs^[2]<D^.E[2]) or (Inputs^[2]>D^.E[3]);
    if Extrapolation then
      Outputs^[2]:=D^.E[5]   {Ausgang 2 auf High-Pegel}
    else
      Outputs^[2]:=D^.E[4];  {Ausgang 2 auf Low-Pegel}
    Interpolation := D^.B[0] = 1;
    {Indizes ix, iy des nächstgelegenen Stützpunkts bestimmen}
    ix := trunc((Inputs^[1] - D^.E[0])/dx) + 1;
    iy := trunc((Inputs^[2] - D^.E[2])/dy) + 1;
    if ix < 1 then ix := 1;
    if iy < 1 then iy := 1;
    if Interpolation then begin
      if ix > nx-1 then ix := nx-1;
      if iy > ny-1 then iy := ny-1;
```

```
      x0 := D^.E[0] + (ix-1)*dx; {Referenzpunkt, x-Koordinate}
      y0 := D^.E[2] + (iy-1)*dy; {Referenzpunkt, y-Koordinate}
      xp := (z[ix+1, iy] - z[ix, iy]) / dx; {Steigung in x-Richtung}
      yp := (z[ix, iy+1] - z[ix, iy]) / dy; {Steigung in y-Richtung}
      Outputs^[1] := z[ix, iy] + (Inputs^[1] - x0)*xp + (Inputs^[2]-y0)*yp;
    end else begin
      if ix > nx then ix := nx;
      if iy > ny then iy := ny;
      Outputs^[1] := z[ix, iy];
    end;
  end;
end;

procedure InitSimulationDLL(D:PParameterStruct;Inputs:PInputArray;
                            Outputs:POutputArray);export stdcall;
var i, j: Integer;
    Datei: text;
begin
  {Erst Parameterdatei einlesen...}
  assign(Datei, D^.D);
  reset(Datei);
  with PUserData(D^.UserDataPtr)^ do begin
    readln(Datei,ny,nx);
    for i:=1 to ny do
      for j:=1 to nx do readln(Datei,z[j, i]);
    dx:=(D^.E[1]-D^.E[0])/(nx-1); {Segmentgröße in x-Richtung errechnen!}
    dy:=(D^.E[3]-D^.E[2])/(ny-1); {Segmentgröße in y-Richtung errechnen!}
  end;
  close(Datei);
  {...und dann einen normalen Simulationsschritt anschließen}
  SimulateDLL(0, D, Inputs, Outputs);
end;

procedure EndSimulationDLL;export stdcall;
begin
  {wird nicht benötigt}
end;

procedure InitUserData(D: PParameterStruct); export stdcall;
begin
  {Speicherplatz für User-Daten anlegen}
  GetMem(D^.UserDataPtr, sizeof(TUserData));
end;

procedure DisposeUserData(D: PParameterStruct); export stdcall;
begin
  {Speicherplatz für User-Daten wieder freigeben}
  FreeMem(D^.UserDataPtr, sizeof(TUserData));
end;

function SetInputChar: PChar; export stdcall;
begin
  {Ersten Eingang mit 'x', zweiten mit 'y' beschriften}
  SetInputChar := 'xy';
end;

function SetOutputChar: PChar; export stdcall;
begin
  {Ersten Ausgang mit 'z', zweiten mit 'E' beschriften}
  SetOutputChar := 'zE';
end;

procedure IsUserDLL32; export stdcall;
begin
end;
```



```
{Exportieren der notwendigen Funktionen und Prozeduren }
exports
  GetParameterStruct, GetDialogEnableStruct, GetNumberOfInputsOutputs,
  CanSimulateDLL, InitSimulationDLL, SimulateDLL,
  InitUserData, DisposeUserData, EndSimulationDLL,
  SetInputChar, SetOutputChar, IsUserDLL32;

begin
  {Weitere Initialisierung der DLL (nicht notwendig).}
end.
```

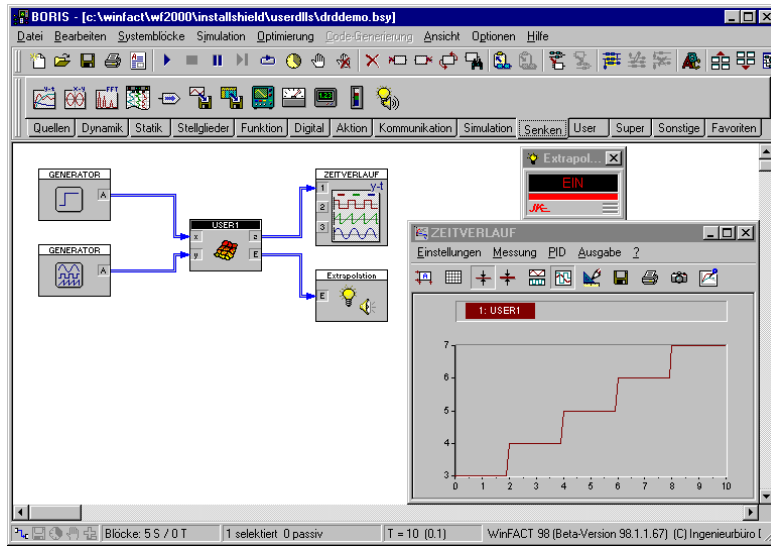
Pascal-Listing für Beispiel 2

Anwendungsbeispiel für Kennfeld-DLL

Es soll ein Kennfeld definiert werden für den Bereich $0 \le x \le 4$, $0 \le y \le 4$, bei dem der Ausgangswert (*z*-Achse) linear von 1 bis 9 ansteigt. Das Kennfeld soll in beiden Richtungen über jeweils 5 Stützpunkte festgelegt sein. Dann hat die zugehörige Kennfeldmatrix folgendes Aussehen:

$$\underline{M} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}.$$

Die entsprechende FWM-Datei befindet sich unter dem Namen DRDDE-MO.FWM im Verzeichnis *UserDLLs*. Es soll der Verlauf der Ausgangsgröße für den Fall simuliert werden, daß die Eingangsgröße *x* konstant 2 ist und die Eingangsgröße *y* linear von 0 auf 5 ansteigt. Es soll dabei keine Interpolation stattfinden. Nachfolgende Grafik zeigt das entsprechende Simulationsergebnis. Die zugehörige Systemdatei befindet sich unter dem Namen DRDDEMO.BSY ebenfalls im Verzeichnis *UserDLLs.*

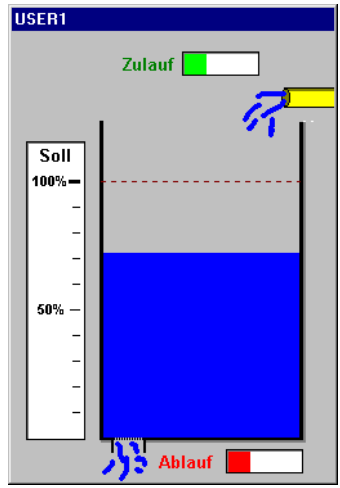


Simulationsergebnis

Beispiel 3: Visualisierung einer Füllstandsregelung

Es soll eine User-DLL erstellt werden, die zur Visualisierung einer Füllstandsregelung dient. Dabei soll beim Starten des ersten Simulationslaufs ein Ausgabefenster erscheinen, in dem die Eingangsgrößen des Blocks (Zulauf, Ablauf, Füllhöhe) visualisiert werden (siehe nachfolgende Grafik). Dieses Ausgabefenster soll erst dann wieder verschwinden, wenn der Block gelöscht wird. Bei der Programmierung soll davon ausgegangen werden, daß alle Eingangsgrößen zwischen 0 und 1 liegen (entsprechend 0 bzw. 100% Zulauf, Ablauf oder Füllhöhe).

Nachfolgendes Listing zeigt die Realisierung der DLL allein auf Basis der Windows-API. Die fertige DLL befindet sich unter dem Namen FUELLSTD.DLL im Verzeichnis *UserDLLs*, der Quelltext FUELLSTD.DPR im Unterverzeichnis *Sources*. Eine wesentlich einfachere Programmierung derartiger Blocktypen läßt sich mit Entwicklungsumgebungen wie DELPHI, VISUAL BASIC etc. erreichen; eine ganze Reihe von in DELPHI 3 realisierten Beispielen (teilweise mit Quelltexten) finden Sie im *UserDLLs*- bzw. *Sources*-Verzeichnis.



Visualisierungsfenster der User-DLL

```
Library FuellStd;

(************************************************)
(*                                              *)
(*        DLL zur Visualisierung einer          *)
(*            Füllstandsregelung                *)
(*         (Beispiel 3 des Handbuchs)           *)
(*                                              *)
(************************************************)

uses WinProcs, WinTypes, Windows, SysUtils, Messages;

{$R FUELLBMP}  {RES-Datei Hintergrund-Bitmap für das Ausgabefenster}


const
  cxBitmap = 250;  (* Breite des Bitmaps *)
  cyBitmap = 350;  (* Höhe des Bitmaps   *)

 {Damit ggfls. auch mehrere DLL-Blöcke dieses Typs gleichzeitig simuliert
  werden können, müssen wir uns für jedes Fenster das Fensterhandle und die
  aktuellen Eingangswerte merken. Außerdem soll jedes Fenster den Namen des
  zugehörigen User-DLL-Blocks im Titel anzeigen. Eleganterweise macht man
  das in einer verketteten Liste. Damit es nicht ganz so kompliziert wird,
  wählen wir hier aber ein statisches Array der Dimension 100 (mehr Blöcke
  wird wohl kaum jemand nehmen!)}
var
  nWindows: integer;  {enthält die Anzahl der geöffneten Fenster}
  LevelWindows: array[1..100] of record
    HW: HWnd;                      {Fensterhandle}
    VPos, VNeg, Height: extended; {aktuelle Eingangswerte des Blocks:
                                   Zulauf, Ablauf, Füllstand}
  end;
```

```
type
  PParameterStruct=^TParameterStruct;
  TParameterStruct=packed record
    NuE : Byte;                       {Anzahl reeller Zahlenwerte}
    NuI : Byte;                       {Anzahl ganzer Zahlenwerte}
    NuB : Byte;                       {Anzahl Schalter}
    E:Array[0..31] of Extended;       {reelle Zahlenwerte}
    I:Array[0..31] of Integer;        {ganze Zahlenwerte}
    B:Array[0..31] of Byte;           {Schalter}
    D:Array[0..255] of char;          {event. Dateiname für weitere Daten.}
    EMin:Array[0..31] of Extended;
    EMax:Array[0..31] of Extended;
    IMin:Array[0..31] of Integer;
    IMax:Array[0..31] of Integer;
    NaE : Array[0..31,0..40] of char; {Namen der rellen Zahlenwerte}
    NaI : Array[0..31,0..40] of char; {Namen der ganzen Zahlenwerte}
    NaB : Array[0..31,0..40] of char; {Namen der Schalter}
    UserDataPtr: Pointer;             {benutzerdef. Daten (nicht benötigt)}
    ParentPtr: Pointer;               {Zeiger auf User-DLL-Block}
    ParentHWnd: HWnd;                 {BORIS-Fensterhandle}
    ParentName: PChar;                {Name des User-DLL-Blocks}
    UserHWindow: HWnd;                {Handle des Ausgabefensters}
    DataFile: text;                   {Text-Datei (hier nicht benötigt)}
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;                { Soll die Eingabe eines Wertes   }
    AllowI: Longint;                { un-/zulässig sein so ist das Bit}
    AllowB: Longint;                { des Allow?-Feldes 0 bzw. 1}
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                         {Anzahl Eingänge}
    Outputs:Byte;                         {Anzahl Ausgänge}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;

  PInputArray = ^TInputArray;
  TInputArray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputArray = packed array[1..50] of extended;


function GetVPos(H: HWnd): extended;
{Diese Funktion liefert für das Fenster mit dem Handle H den zugehörigen
 Parameter VPos, d. h. den Zulauf zurück}
var i: integer;
begin
  {Fensterliste nach passendem Fenster durchsuchen}
  for i:=1 to nWindows do if LevelWindows[i].HW = H then
    GetVPos := LevelWindows[i].VPos;
end;

function GetVNeg(H: HWnd): extended;
{Diese Funktion liefert für das Fenster mit dem Handle H den zugehörigen
 Parameter VNeg, d. h. den Zulauf zurück}
var i: integer;
begin
  {Fensterliste nach passendem Fenster durchsuchen}
  for i:=1 to nWindows do if LevelWindows[i].HW = H then
    GetVNeg := LevelWindows[i].VNeg;
```



```
end;

function GetHeight(H: HWnd): extended;
{Diese Funktion liefert für das Fenster mit dem Handle H den zugehörigen
 Parameter Height, d. h. die Füllhöhe zurück}
var i: integer;
begin
  {Fensterliste nach passendem Fenster durchsuchen}
  for i:=1 to nWindows do if LevelWindows[i].HW = H then
    GetHeight := LevelWindows[i].Height;
end;

procedure SetInputs(H: HWnd; VP, VN, Hgt: extended);
{Setzt in der Fensterliste die Eingangswerte des Fensters mit Handle H}
var i: integer;
begin
  for i:=1 to nWindows do if LevelWindows[i].HW = H then begin
    LevelWindows[i].VPos := VP;     {Zulauf}
    LevelWindows[i].VNeg := VN;     {Ablauf}
    LevelWindows[i].Height := Hgt;  {Füllhöhe}
  end;
end;

procedure Paint(DC: HDC; VP, VN, Hgt: extended);
(* Übernimmt die eigentliche Zeichnung des Ausgabefensters *)
var MemDC: HDC;
    RedBrush, BlueBrush, GreenBrush: HBrush;
    HBM: HBitmap;
    BlockBitmap: TBitmap;
    Rect: TRect;
begin
  MemDC := CreateCompatibleDC(DC);
  HBM := LoadBitmap(HInstance, 'BEHAELTER_BITMAP');
  SelectObject(MemDC, HBM);
  GetObject(HBM, SizeOf(BlockBitmap), @BlockBitmap);
  RedBrush := CreateSolidBrush(RGB(255, 0, 0));
  BlueBrush := CreateSolidBrush(RGB(0, 0, 255));
  GreenBrush := CreateSolidBrush(RGB(0, 255, 0));
  {Zulauf}
  with Rect do begin
    Left := 134;
    Right := Left + round(VP*57);
    Top := 17;
    Bottom := 33;
    FillRect(MemDC, Rect, GreenBrush);
  end;
  {Ablauf}
  with Rect do begin
    Left := 168;
    Right := Left + round(VN*57);
    Top := 327;
    Bottom := 343;
    FillRect(MemDC, Rect, RedBrush);
  end;
  {Füllstand}
  with Rect do begin
    Left := 71;
    Right := 223;
    Bottom := 316;
    Top := Bottom - round(Hgt*200);
    FillRect(MemDC, Rect, BlueBrush);
  end;
  BitBlt(DC, 0, 0, BlockBitmap.bmwidth, BlockBitmap.bmheight, MemDC, 0, 0,
         SrcCopy);
  DeleteObject(RedBrush);
```

```
  DeleteObject(GreenBrush);
  DeleteObject(BlueBrush);
  DeleteDC(MemDC);
  DeleteObject(HBM);
end;

function WndFunc(Wnd: HWnd; Msg, wParam: word; lParam: longint): longint;
                                                                    stdcall;
{Fensterfunktion des Ausgabefensters}
var PaintStruct: TPaintStruct;
    DC: HDC;
    i, Index: integer;
begin
  case Msg of
    wm_Paint: begin
                DC := BeginPaint(Wnd, PaintStruct);
                Paint(DC, GetVPos(Wnd), GetVNeg(Wnd), GetHeight(Wnd));
                EndPaint(Wnd, PaintStruct);
              end;
    wm_Destroy: begin
                  {Das Fenster soll zerstört werden. Wir müssen es also
                   aus der Fensterliste löschen und alle anderen Einträge
                   um eins nach vorn verschieben}
                  Index := 0;
                  repeat inc(Index) until LevelWindows[Index].HW = Wnd;
                  for i:=Index+1 to nWindows do
                    LevelWindows[i-1] := LevelWindows[i];
                  dec(nWindows);
                end;
  end;
  WndFunc := DefWindowProc(Wnd, Msg, wParam, lParam);
end;

const
  WindowClass: TWndClass = (
    style: 0;
    lpfnWndProc: @WndFunc;
    cbClsExtra: 0;
    cbWndExtra: 0;
    hInstance: 0;
    hIcon: 0;
    hCursor: 0;
    hbrBackground: 1;
    lpszMenuName: nil;
    lpszClassName: 'LevelWindow');

procedure GetParameterStruct(D:PParameterStruct); export stdcall;
begin
  (* keine Parameter erforderlich ! *)
  D^.NuE := 0;
  D^.NuI := 0;
  D^.NuB := 0;
end;

procedure GetDialogEnableStruct(D:PDialogEnableStruct;D2:PParameterStruct);
                                                          export stdcall;
begin
  (* wird hier nicht benötigt, da kein Dialog erforderlich *)
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
                                                        export stdcall;
begin
  D^.Inputs:=3;    {Unser Block hat drei Eingänge...}
  D^.Outputs:=0;   {            und keinen Ausgang  }
```



```
  {Namen der iIngänge}
  StrCopy(D^.NameI[0], 'Zulauf');
  StrCopy(D^.NameI[1], 'Ablauf');
  StrCopy(D^.NameI[2], 'Füllstand');
end;

procedure InitUserDLL(D:PParameterStruct); export stdcall;
{Diese Prozedur wird vom User-DLL-Block aufgerufen, wenn er initialisiert
 wird. Wir benötigen ein Flag, an dem wir später erkennen können,
 ob das Ausgabefenster schon existiert. Dazu nehmen wir den ersten
 Integer-Parameter D^.I[0]. Im Prinzip könnten wir das Ausgabefenster auch
 schon hier erzeugen; wir wollen aber, daß es erst bei Start der ersten
 Simulation erscheint; dadurch wird die ganze Sache etwas komplizierter!}
begin
  D^.I[0] := 0;
end;

procedure DisposeUserDLL(D:PParameterStruct); export stdcall;
{Diese Prozedur wird vom User-DLL-Block aufgerufen, wenn der Block gelöscht
 wird oder eine andere DLL für den Block geladen wird. In diesen Fällen
 müssen wir das Ausgabefenster löschen}
begin
  DestroyWindow(D^.UserHWindow);
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
begin
  CanSimulateDLL:=1;  {Simulation immer möglich}
end;

procedure InitSimulationDLL(D:PParameterStruct;Inputs:PInputArray;
                            Outputs:POutputArray); export stdcall;
var Rect: TRect;
    x, y: integer;
begin
  with Rect do begin
    Top := 0;
    Bottom := cyBitmap;
    Left := 0;
    Right := cxBitmap;
  end;
  (* Fenstergröße an Bitmapgröße anpassen! *)
  AdjustWindowRect(Rect, ws_Popup or ws_Caption, false);
  {Falls noch kein Ausgabefenster existiert, wird es jetzt angelegt...}
  if D^.I[0] = 0 then begin
    D^.I[0] := 1; {kennzeichnen, daß Fenster jetzt vorhanden}
    {Damit bei mehreren Fenster nicht alle aufeinander liegen, verschieben
     wir jedes um 50 Pixel nach links oben}
    x := 350 + nWindows*50;
    y := 70 - nWindows*50;
    {Es ist soweit: Das Fenster wird erzeugt...}
    D^.UserHWindow := CreateWindowEx(ws_ex_TopMost, 'LevelWindow', '',
                              ws_Popup or ws_Caption or ws_MinimizeBox,
                              x, y, Rect.Right-Rect.Left, Rect.Bottom-
                              Rect.Top, D^.ParentHWnd, 0, HInstance, nil);
    {...angezeigt...}
    ShowWindow(D^.UserHWindow, sw_Show);
    {...und in die Fensterliste eingetragen!}
    inc(nWindows);
    LevelWindows[nWindows].HW := D^.UserHWindow;
  end;
  {Wir wollen dem Fenster den Namen des User-DLL-Blocks geben}
  SetWindowText(D^.UserHWindow, D^.ParentName);
end;
```

```
procedure SimulateDLL(T:Extended;D:PParameterStruct;Inputs:PInputArray;
                      Outputs:POutputArray);export stdcall;
var DC: HDC;
begin
  {Inputs in Fensterliste eintragen...}
  SetInputs(D^.UserHWindow, Inputs^[1], Inputs^[2], Inputs^[3]);
  {...und Fensterinhalt neu zeichnen}
  DC := GetDC(D^.UserHWindow);
  Paint(DC, GetVPos(D^.UserHWindow), GetVNeg(D^.UserHWindow),
        GetHeight(D^.UserHWindow));
  ReleaseDC(D^.UserHWindow, DC);
end;

procedure EndSimulationDLL; export stdcall;
begin
  (* hier nicht erforderlich *)
end;

function SetInputChar: PChar; export stdcall;
begin
  {Ersten Eingang mit '+', zweiten mit '-' und dritten mit 'H' beschriften}
  SetInputChar := '+-H';
end;

procedure ShowWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Anzeigefenster in Normalgröße anzeigen}
  ShowWindow(D^.UserHWindow, sw_Normal);
end;

procedure HideWindowDLL(D: PParameterStruct); export stdcall;
begin
  {Anzeigefenster zum Symbol verkleinern}
  ShowWindow(D^.UserHWindow, sw_Minimize);
end;

procedure IsUserDLL32; export stdcall;
begin
end;

exports
  InitUserDLL, DisposeUserDLL, GetParameterStruct,
  GetDialogEnableStruct, GetNumberOfInputsOutputs,
  CanSimulateDLL, InitSimulationDLL, SimulateDLL,
  EndSimulationDLL, SetInputChar, ShowWindowDLL,
  HideWindowDLL, IsUserDLL32;

begin
  (* Fensterklasse registrieren *)
  WindowClass.hInstance := HInstance;
  WindowClass.hIcon := LoadIcon(0, idi_Application);
  WindowClass.hCursor := LoadCursor(0, idc_Arrow);
  RegisterClass(WindowClass);
  nWindows := 0;
end.
```

Pascal-Listing zu Beispiel 3

Vorlagendatei DLLTEMPL.PAS

```
Library DLLtempl;

{#COMMENT}

uses WinProcs, SysUtils, WinTypes;

type
  PParameterStruct=^TParameterStruct;
  TParameterStruct=packed record
    NuE : Byte;                       {Anzahl reeller Zahlenwerte}
    NuI : Byte;                       {Anzahl ganzer Zahlenwerte}
    NuB : Byte;                       {Anzahl Schalter}
    E:Array[0..31] of Extended;       {reelle Zahlenwerte}
    I:Array[0..31] of Integer;        {ganze Zahlenwerte}
    B:Array[0..31] of Byte;           {Schalter}
    D:Array[0..255] of char;          {event. Dateiname für weitere Daten.}
    EMin:Array[0..31] of Extended;    {untere Eingabegrenze für jeden
                                       reellen Zahlenwert}
    EMax:Array[0..31] of Extended;    {obere Eingabegrenze für jeden rellen
                                       Zahlenwert}
    IMin:Array[0..31] of Integer;     {untere Eingabegrenze für jeden
                                       ganzzahligen Zahlenwert}
    IMax:Array[0..31] of Integer;     {obere Eingabegrenze für jeden
                                       ganzzahligen Zahlenwert}
    NaE : Array[0..31,0..40] of char;   {Namen der reellen Zahlenwerte}
    NaI : Array[0..31,0..40] of char;   {Namen der ganzen Zahlenwerte}
    NaB : Array[0..31,0..40] of char;   {Namen der Schalter}
    UserDataPtr: Pointer;               {Zeiger auf weitere Blockvariablen}
    ParentPtr: Pointer;                 {Zeiger auf User-DLL-Block}
    ParentHWnd: HWnd;                   {BORIS-Fensterhandle}
    ParentName: PChar;                  {Name des User-DLL-Blocks}
    UserHWindow: HWnd;                  {Benutzerdef. Fensterhandle, z. B.
                                         für Ausgabefenster}
    DataFile: text;                     {Textdatei für universelle Zwecke}
  end;

  PDialogEnableStruct=^TDialogEnableStruct;
  TDialogEnablestruct=packed record
    AllowE: Longint;                    { Soll die Eingabe eines Wertes   }
    AllowI: Longint;                    { un-/zulässig sein so ist das Bit}
    AllowB: Longint;                    { des Allow?-Feldes 0 bzw. 1}
    AllowD: Byte;
  end;

  PNumberOfInputsOutputs=^TNumberOfInputsOutputs;
  TNumberOfInputsOutputs=packed record
    Inputs :Byte;                         {Anzahl Eingänge}
    Outputs:Byte;                         {Anzahl Ausgänge}
    NameI : Array[0..49,0..40] of char;
    NameO : Array[0..49,0..40] of char;
  end;
```

```
  PInputArray = ^TInputArray;
  TInputArray = packed array[1..50] of extended;
  POutputArray = ^TOutputArray;
  TOutputArray = packed array[1..50] of extended;

  {#USER_DATA}

procedure GetParameterStruct(D:PParameterStruct);export stdcall;
begin
  {#PARA}
end;

procedure GetDialogEnableStruct(D:PDialogEnableStruct;
                                D2:PParameterStruct);export stdcall;
begin
  {Alle Dialogelemente sollen jederzeit zugänglich sein!}
  D^.AllowE:=$FFFFFFFF;
  D^.AllowB:=$FFFFFFFF;
  D^.AllowI:=$FFFFFFFF;
  D^.AllowD:= 1;
end;

procedure GetNumberOfInputsOutputs(D:PNumberOfInputsOutputs);
                                                        export stdcall;
begin
  {#IN_OUT}
end;

function CanSimulateDLL(D:PParameterStruct):Integer; export stdcall;
begin
  CanSimulateDLL := 1;
end;

procedure SimulateDLL2(T, DeltaT:Extended;D:PParameterStruct;
                Inputs:PInputArray;Outputs:POutputArray);export stdcall;
begin
  {#SIM}
end;

procedure InitSimulationDLL(D:PParameterStruct;
                Inputs:PInputArray;Outputs:POutputArray);export stdcall;
begin
  {#INIT_SIM}
end;

procedure EndSimulationDLL2(D: PParameterStruct);export stdcall;
begin
  {#END_SIM}
end;

procedure InitUserData(D: PParameterStruct); export stdcall;
begin
  {kann evtl. statt InitUserDLL benutzt werden, wenn die Daten nur während
der
   Simulation selbst benötigt werden}
end;
```



```
procedure DisposeUserData(D: PParameterStruct); export stdcall;
begin
  {kann evtl. statt DisposeUserDLL benutzt werden, wenn die Daten nur
während der
   Simulation selbst benötigt werden}
end;

procedure InitUserDLL(D: PParameterStruct); export stdcall;
begin
  GetMem(D^.UserDataPtr, Sizeof(TUserData));
end;

procedure DisposeUserDLL(D: PParameterStruct); export stdcall;
begin
  FreeMem(D^.UserDataPtr, Sizeof(TUserData));
end;

function SetInputChar: PChar; export stdcall;
begin
  {Diese Funktion wird nur gefüllt und exportiert, wenn im User-DLL-
   Experten bei der Beschriftung der Blockein- und -ausgänge
   "Individuelle Beschriftung" angegeben wird; sonst bleibt sie leer!}
  {#SET_INPUT_CHAR}
end;

function SetOutputChar: PChar; export stdcall;
begin
  {Diese Funktion wird nur gefüllt und exportiert, wenn im User-DLL-
   Experten bei der Beschriftung der Blockein- und -ausgänge
   "Individuelle Beschriftung" angegeben wird; sonst bleibt sie leer!}
  {#SET_OUTPUT_CHAR}
end;

procedure IsUserDLL32; export stdcall;
begin
  {Wird benötigt, damit BORIS die DLL akzeptiert}
end;

{Exportieren der notwendigen Funktionen und Prozeduren }
exports
  {#EXP_SET_INPUT_CHAR}
  {#EXP_SET_OUTPUT_CHAR}
  GetParameterStruct,
  GetDialogEnableStruct,
  GetNumberOfInputsOutputs,
  CanSimulateDLL,
  InitSimulationDLL,
  SimulateDLL2,
  InitUserDLL,
  DisposeUserDLL,
  InitUserData,
  DisposeUserData,
  EndSimulationDLL2,
  IsUserDLL32;
```

```
begin
  {Weitere Initialisierung der DLL}
end.
```

Vorlagendatei DLLTEMPL.C

```
#include <vcl.h>
#include <windows.h>
#include <string.h>
#include <math.h>
#include <malloc.h>
#pragma hdrstop
#pragma pack(1)

/*
{#COMMENT}
*/

typedef struct{
    char NuE;                //Anzahl reeller Zahlenwerte
    char NuI;                //Anzahl ganzer Zahlenwerte
    char NuB;                //Anzahl Schalter
    long double E[32];       //reelle Zahlenwerte
    long I[32];              //ganze Zahlenwerte
    char B[32];              //Schalter
    char D[256];             //event. Dateiname für weitere Daten.
    long double EMin[32];    //untere Eingabegrenze für jeden reellen
                               Zahlenwert
    long double EMax[32];    //obere Eingabegrenze für jeden reellen
                               Zahlenwert
    long IMin[32];           //untere Eingabegrenze für jeden
                               ganzzahligen Zahlenwert
    long IMax[32];           //obere Eingabegrenze für jeden
                               ganzzahligen Zahlenwert
    char NaE[32][41];        //Namen der reellen Zahlenwerte
    char NaI[32][41];        //Namen der ganzen Zahlenwerte
    char NaB[32][41];        //Namen der Schalter
    void *UserDataPtr;       //Zeiger auf User-Daten
    void *ParentPtr;         //Zeiger auf User-Block
    unsigned int ParentHWnd; //BORIS-Fensterhandle
    char *ParentName;        //Name des User-Blocks
    unsigned int UserHWindow;//Benutzerdef. Fensterhandle, z. B. für
                               Ausgabefenster
    void *DataFile;          //Datei für Benutzerzwecke
  } TParameterStruct, FAR *PParameterStruct;

typedef struct {
    long AllowE;            //Soll die Eingabe eines Wertes
    long AllowI;            //un-/zulässig sein so ist das Bit
    long AllowB;            //des Allow?-Feldes 0 bzw. 1
    char AllowD;
  } TDialogEnableStruct, FAR *PDialogEnableStruct;
```



```
typedef struct {
    char Inputs;                //Anzahl Eingänge
    char Outputs;               //Anzahl Ausgänge
    char NameI[50][41];
    char NameO[50][41];
  } TNumberOfInputsOutputs, FAR *PNumberOfInputsOutputs;

typedef long double TInputArray[50];
typedef long double TOutputArray[50];

#pragma pack()

{#USER_DATA}

extern "C"
{
  void  _export _stdcall GetParameterStruct(PParameterStruct);
  void  _export _stdcall InitUserDLL(PParameterStruct);
  void  _export _stdcall DisposeUserDLL(PParameterStruct);
  void  _export _stdcall InitUserData(PParameterStruct);
  void  _export _stdcall DisposeUserData(PParameterStruct);
  void  _export _stdcall GetDialogEnableStruct(PDialogEnableStruct,
                                               PParameterStruct);
  void  _export _stdcall GetNumberOfInputsOutputs(PNumberOfInputsOutputs
                                                  D);
  int   _export _stdcall CanSimulateDLL(PParameterStruct D);
  void  _export _stdcall InitSimulationDLL(PParameterStruct D,
                         TInputArray FAR Inputs,TOutputArray FAR Outputs);
  void  _export _stdcall SimulateDLL(long double T, PParameterStruct D,
                         TInputArray FAR Inputs,TOutputArray FAR Outputs);
  void  _export _stdcall SimulateDLL2(long double T,long double DeltaT,
                           PParameterStruct D,TInputArray FAR Inputs,
                           TOutputArray FAR Outputs);
  void  _export _stdcall EndSimulationDLL(void);
  void  _export _stdcall EndSimulationDLL2(PParameterStruct D);
  void  _export _stdcall SetEnhancedInformation(long double DeltaT,
                                  long double T, PParameterStruct D);
  void  _export _stdcall IsUserDLL32(void);
  {#EXP_SET_INPUT_CHAR}
  {#EXP_SET_OUTPUT_CHAR}
}

void _export _stdcall  GetParameterStruct(PParameterStruct D)
{
{#PARA}
}

void _export _stdcall  InitUserDLL(PParameterStruct D)
{
  D->UserDataPtr = (PUserData)malloc(sizeof(TUserData));
}

void _export _stdcall  DisposeUserDLL(PParameterStruct D)
{
```

```
  free(D->UserDataPtr);
}

void _export _stdcall  InitUserData(PParameterStruct D)
{
}

void _export _stdcall  DisposeUserData(PParameterStruct D)
{
}


void _export _stdcall GetDialogEnableStruct(PDialogEnableStruct D,
                                                    PParameterStruct D2)
{
//Alle Dialogelemente sollen jederzeit zugänglich sein
  D->AllowE=0xFFFFFFFF;
  D->AllowI=0xFFFFFFFF;
  D->AllowB=0xFFFFFFFF;
  D->AllowD=0x01;
}


void _export _stdcall  GetNumberOfInputsOutputs(PNumberOfInputsOutputs D)
{
{#IN_OUT}
}


int _export _stdcall  CanSimulateDLL(PParameterStruct D)
{ //Simulation immer zulässig!
  return 1;
}

void _export _stdcall SimulateDLL(long double T,PParameterStruct D,
                         TInputArray FAR Inputs,TOutputArray FAR Outputs)
{
  //Deklaration und Export dieser Prozedur ist aus Kompatibilitätsgründen
  //zu alten Versionen erforderlich!
}

void _export _stdcall SimulateDLL2(long double T,long double DeltaT,
                         PParameterStruct D,TInputArray FAR Inputs,
                         TOutputArray FAR Outputs)
{
{#SIM}
}


void _export _stdcall InitSimulationDLL(PParameterStruct D,
                         TInputArray FAR Inputs,TOutputArray FAR Outputs)
{
{#INIT_SIM}
}

void _export _stdcall EndSimulationDLL(void)
```



```
{
  //Deklaration und Export dieser Prozedur ist aus Kompatibilitätsgründen
  //zu alten Versionen erforderlich!
}

void _export _stdcall EndSimulationDLL2(PParameterStruct D)
{
{#END_SIM}
}

void _export _stdcall SetEnhancedInformation(long double DeltaT,
                                long double T, PParameterStruct D)
{
// Wird vom User-DLL-Experten nicht benutzt; kann ggfls. vom Anwender
// selbst gefüllt werden
}

char * _export _stdcall SetInputChar(void)
{
  // Diese Funktion wird nur gefüllt und exportiert, wenn im User-DLL-
  // Experten bei der Beschriftung
  // der Blockein- und -ausgänge "Individuelle Beschriftung" angegeben
  // wird; sonst bleibt sie leer!
{#SET_INPUT_CHAR}
}

char * _export _stdcall SetOutputChar(void)
{
  // Diese Funktion wird nur gefüllt und exportiert, wenn im User-DLL-
  // Experten bei der Beschriftung
  // der Blockein- und -ausgänge "Individuelle Beschriftung" angegeben
  // wird; sonst bleibt sie leer!
{#SET_OUTPUT_CHAR}
}

void _export _stdcall IsUserDLL32(void)
{
}

//---------------------------------------------------------------------------
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}
//---------------------------------------------------------------------------
```