Rätselhafte Argumente

Programmierung benutzerdefinierter Funktionen für InterBase – Teil 2

von Karsten Strobel

Es ist farblos, hat keinen Geruch und einen beschränkten Horizont. Es ist grobkörnig und man verwendet es oft, um sich etwas zu merken. Es ist so klein, dass es fast niemand wirklich schon mal gesehen hat, aber trotzdem wird es massenhaft benutzt. Was ist das?

... Hmmm... Ein Integer Parameter?

Im ersten Teil dieses Artikels hatte ich die Grundregeln für das Schreiben eigener benutzerdefinierter Funktionen – verbunden mit vielen Sicherheitshinweisen – dargestellt und einige Beispiele für die UDF-Programmierung mit einfachen Parametertypen, die per Referenz an die Funktion übergeben werden, ausgearbeitet. In diesem zweiten Teil geht es um die InterBase-Datentypen, für deren Behandlung etwas mehr Aufwand nötig wird, wenn man sie als Funktionsparameter deklariert. Diese etwas schwierigeren Kandidaten sind VARCHAR, DATE, TIME, TIMESTAMP und BLOB. Sie kommen jeweils verpackt in eine besondere Struktur bei der Funktion an.

Quellcode

Den Quellcode finden Sie auf der aktuellen Profi-CD sowie auf unserer homepage unter

www.derentwickler.de

Um mit diesen von InterBase vorgegebenen Strukturen mit Delphi korrekt umgehen zu können, müssen Sie kompatible Records typisieren. Listing 1 zeigt die Delphi-Unit *udf_def.pas* mit den nötigen Definitionen, die Sie in das im letzten Heft begonnene Projekt „MeineUDFs.dpr“ mit *uses* einbinden können. Alle Quellcodes findet man auch auf der Profi-CD und im Web.

Maß halten

Ganz einfach ist die für VARCHAR-Parameter vorgesehene Struktur *TVarChar* aufgebaut: Hier gibt das Längenwort *len* die Anzahl der Zeichen im darauf folgenden *text* an. Man sollte sich nicht darauf verlassen, dass *text* mit einem Null-Zeichen abgeschlossen wird. Meistens ist es einfacher, statt eines VARCHAR-Parameters den Pseudo-Typ CSTRING, den wir im ersten Teil schon kennen gelernt haben, zu deklarieren, denn dieser lässt sich innerhalb der Funktion als nullterminierter String handhaben. Ich möchte daher auf VARCHAR gar nicht weiter eingehen. Wenn Sie diesen Typ dennoch verwenden wollen, nutzen Sie für die Deklaration eines solchen UDF-Arguments den Zeigertyp *PVarChar*.

Etwas komplizierter ist der Umgang mit den verschiedenen Datum- und Zeit-Feldtypen. Bei InterBase V5 gab es nur den DATE-Feldtyp, der aber außer einem Datum auch eine Uhrzeit speicherte. Ab InterBase 6 wird dieser Feldtyp konsequenterweise als TIMESTAMP bezeichnet und es gibt endlich Typen für reine Datums- und reine Zeitwerte, nämlich DATE und TIME. Die Bedeutung des reservierten Wortes DATE hat sich also zwischen diesen Versionen geändert. Um die Migration von V5 nach V6 zu vereinfachen, unterstützt InterBase V6 den SQL-Dialekt 1 als Kompatibilitätsmodus. Listing 1 enthält deshalb die Typdefinitionen *PDate_Dialect_1* und *PDate_Dialect_3*. Erstere kann auch für InterBase V5 verwendet werden.

Ein TIMESTAMP-Parameter wird an eine UDF als Zeiger auf eine 64-Bit-Struktur übergeben, die Datum und Zeit in zwei 32-Bit Werten enthält. Reine Datums- oder Zeitwerte kommen als Zeiger auf ein einziges 32-Bit-Wort, können prinzipiell also wie Integers behandelt werden. Bei InterBase V6 steht die EXTRACT()-Funktion zur Verfügung, mit der viele Aufgaben, für die früher UDFs erforderlich waren, in SQL erledigt werden können.

Ein sinnvolles Beispiel für eine dennoch nützliche UDF ist die Ermittlung der Kalenderwoche aus einem Datum. Wie für die meisten Dinge im Leben gibt es auch hier eine Norm, nämlich die ISO 8601, die die Berechnung dieser Zahl aus einem Datum definiert. Der erste Januar eines Jahres liegt nämlich nicht immer in der ersten Kalenderwoche. Je nachdem, auf welchen Wochentag dieses Datum fällt, kann dieser Tag auch noch zur letzten Kalenderwoche des abgelaufenen Jahres gerechnet werden. Daher muss auch ermittelt werden können, in welches Kalenderjahr die Kalenderwoche eines Datums fällt. Listing 2 zeigt eine

Lösung für dieses Problem, wobei der Code für die eigentliche Berechnung aus dem *Project Jedi* [1] stammt.

Im ersten Parameter wird das fragliche Datum übergeben. Der zweite Parameter wählt zwischen der Kalenderwoche und dem Kalenderjahr als Funktionsergebnis aus. Hier ein Beispiel für die Anwendung der Funktion:

```
SELECT 'KW' || ISO_WEEK_NUMBER('01-JAN-2000',
1) || '/' || ISO_WEEK_NUMBER('01-JAN-2000', 2) FROM
RDB$DATABASE

/*
 Ergebnis: KW52/1999
*/
```

Ich habe den Datumsparameter dieser Funktion als TIMESTAMP definiert, obwohl eigentlich nur DATE erforderlich wäre. TIMESTAMP verhält sich aber bei Übergabe von Strings großzügiger und castet im obigen Beispiel automatisch, was bei DATE zu Problemen führen kann. Daher, und weil der Zeitanteil nicht weiter stört und auch der Typbezeichner eindeutiger ist, gibt man beim Programmieren von UDFs diesem Parametertyp gegenüber DATE oft den Vorzug.

Die UDF verwendet *isc_decode_date()*, um den erhaltenen Zeitstempel, der nach anderen Vorschriften als Delphis *TDateTime* kodiert ist, in seine Bestandteile zu zerlegen. *isc_decode_date()* ist eine Hilfsfunktion, die in der Clientbibliothek *gds32.dll* implementiert ist und rein lokal ausgeführt wird, also keinen Serveraufruf verursacht. Im Initialisierungsteil von *udf_def.pas* wird die DLL geladen und die Funktionsadresse ermittelt. Der InterBase-Zeitstempel wird in eine Struktur vom Typ *tm* entpackt, mit deren Mitgliedern man dann bequem weiterrechnen kann. Das Pendant mit umgekehrter Arbeitsrichtung ist *isc_encode_date()* und kann zum Beispiel zum Einpacken eines Funktionsergebnisses verwendet werden, wenn eine UDF ein Datum zurückgeben soll.

Daten häppchenweise

Auch BLOBs (binary large objects) erfordern eine besondere Behandlung, wenn man sie in UDFs verarbeiten will. Der Inhalt von BLOBs sind häufig Textdaten, aber anders als String-Datentypen kann InterBase diese Daten nicht einfach in den Speicher laden und einen Zeiger auf den Anfang übergeben, denn der Umfang kann theoretisch enorm groß sein und die dynamische Speicherverwaltung in die Knie zwingen. Daher erhält die UDF für einen BLOB-Parameter eine besondere Struktur, in der auch zwei Callback-Adressen enthalten sind, mit deren Hilfe die externe Funktion den BLOB-Inhalt segmentweise laden oder speichern kann. Listing 1 enthält bereits den *TBlob*-Recordtyp und die formale Deklaration der Callback-Funktionen *TBlobGetSegment* und *TBlobPutSegment*. Auskunft über Größe und Aufbau des BLOBs liefern die Felder *NumberSegments*, *MaxSegLen* und *TotalSize*.

Listing 1

```
unit udf_def;

interface

uses Windows, Sysutils;

type
 PVarChar = ^TVarChar;
 TVarChar = packed record
  len: word;
  text: array[0..0] of char;
 end;

 { Interbase Date/Time Record }
 TISC_QUAD = packed record //64 bit structure
  isc_quad_high: Integer; // Date
  isc_quad_low: Cardinal; // Time
 end;
 PISC_QUAD = ^TISC_Quad;

 PTimestamp = PISC_QUAD;
 PDate_Dialect_1 = PTimestamp;
 PDate_Dialect_3 = ^Integer;
 PTime_Dialect_3 = ^Cardinal;

 { C Date/Time Structure }
 tm = packed record
  tm_sec: integer; // Seconds
  tm_min: integer; // Minutes
  tm_hour: integer; // Hour (0—23)
  tm_mday: integer; // Day of month (1—31)
  tm_mon: integer; // Month (0—11)
  tm_year: integer; // Year (calendar year minus 1900)
  tm_wday: integer; // Weekday (0—6) Sunday = 0)
  tm_yday: integer; // Day of year (0—365)
  tm_isdst: integer; // 0 if daylight savings time is not in
                                                effect)
 end;
 ptm = ^tm;

 TISCEncodeDate = procedure (tm_date: ptm; ib_date:
                                  PISC_QUAD); stdcall;
 TISCDecodeDate = procedure (ib_date: PISC_QUAD;
                                  tm_date: ptm); stdcall;

 //=== BLOB control structure ===
 //Possible results of BlobGetSegment are:
 //1=Complete segment returned, 0=End of blob (no data
                  returned), -1=current segment is incomplete
 TBlobGetSegment = function (AHandle: pointer; const
                                                ABuffer: PChar;
          ABufSize: Word; var AResultSize: word):
                                                smallint; cdecl;
 TBlobPutSegment = procedure(AHandle: pointer; const
                  ABuffer: PChar; ABufSize: Word); cdecl;

 PBlob = ^TBlob;
 TBlob = packed record
  BlobGetSegment: TBlobGetSegment;
  BlobHandle: pointer;
  NumberSegments,
  MaxSegLen,
  TotalSize: Integer;
  BlobPutSegment: TBlobPutSegment;
 end;
var
 LibHandle: THandle;
 isc_encode_date: TISCEncodeDate;
 isc_decode_date: TISCDecodeDate;

implementation
initialization
 LibHandle := LoadLibrary('gds32.dll');
 if LibHandle < 32 then
  raise Exception.Create('Error initializing interbase
                                                library.');

 @isc_decode_date := GetProc  Address(LibHandle,
'isc_decode_date');
 if (not Assigned(isc_decode_date)) then
  raise Exception.Create('Unable to locate
isc_decode_date');

 @isc_encode_date := GetProc  Address(LibHandle,
'isc_encode_date');
 if (not Assigned(isc_encode_date)) then
  raise Exception.Create('Unable to locate
isc_encode_date');

finalization
 FreeLibrary(LibHandle);
end.
```

Listing 2

```
function ISOWeekNumber(const ADate: PTimestamp; var AResultMode: Integer): Integer; cdecl; export;
{
DECLARE EXTERNAL FUNCTION ISO_WEEK_NUMBER
TIMESTAMP, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'ISO_WEEK_NUMBER' MODULE_NAME 'MeineUDFs';
}

  // DayOfWeek function returns integer 1..7 equivalent to Sunday..Saturday.
  // ISO 8601 weeks start with Monday and the first week of a year is the one which
  // includes the first Thursday - Fiddle takes care of all this }
  function ISOWeekNumber(AYear, AMonth, ADay: Word; var AYearOfWeekNumber: Integer): Integer;
  const
   Fiddle: array [1..7] of Byte = (6, 7, 8, 9, 10, 4, 5);
  var
   Present, StartOfYear: TDateTime;
   FirstDayOfYear, WeekNumber, NumberOfDays: Integer;
  begin
   Present := EncodeDate(AYear, AMonth, ADay);
   StartOfYear := EncodeDate(AYear, 1, 1); // encode 1st Jan of the year
   // find what day of week 1st Jan is, then add days according to rule
   FirstDayOfYear := Fiddle[DayOfWeek(StartOfYear)];
   // calc number of days since beginning of year + additional according to rule
   NumberOfDays := Trunc(Present - StartOfYear) + FirstDayOfYear;
   // calc number of weeks
   WeekNumber := NumberOfDays div 7;
   if WeekNumber = 0 then
    // see if previous year end was week 52 or 53
    Result := ISOWeekNumber(AYear-1, 12, 31, AYearOfWeekNumber)
   else
   begin
    AYearOfWeekNumber := AYear;
    Result := WeekNumber;
    if WeekNumber = 53 then
    begin
     // if 31st December less than Thursday then must be week 01 of next year
     if DayOfWeek(EncodeDate(AYear, 12, 31)) < 5 then
     begin
      AYearOfWeekNumber := AYear + 1;
      Result := 1;
     end;
    end;
   end;
  end;

var
 tm_date: tm;
 year_of_week: Integer;
begin
 isc_decode_date(ADate, @tm_date);
 with tm_date do
  Result := ISOWeekNumber(tm_year+1900, tm_mon+1, tm_mday, year_of_week);
 if AResultMode = 2 then Result := year_of_week;
end;
```

Listing 3

```
function BlobWordCount(const ABlob: PBlob): Integer; cdecl;
{
DECLARE EXTERNAL FUNCTION BLOB_WORD_COUNT
BLOB
RETURNS INTEGER BY VALUE
ENTRY_POINT 'BLOB_WORD_COUNT' MODULE_NAME 'MeineUDFs';
}
var
 s: string;
 SizeLeft, i: Integer;
 ResultSize: word;
 in_word: boolean;
begin
 Result := 0;
 if Assigned(ABlob) and Assigned(ABlob^.BlobHandle) then
  with ABlob^ do
  begin
   SizeLeft := TotalSize;
   in_word := false;
   while SizeLeft > 0 do
   begin
    SetLength(s, MaxSegLen);
    if BlobGetSegment(BlobHandle, PChar(s),  MaxSegLen, ResultSize) <> 0 then
    begin
     Dec(SizeLeft, ResultSize);
     if ResultSize <> MaxSegLen then SetLength (s, ResultSize);
     for i := 1 to ResultSize do
      if not (s[i] in [#9,#10,#13,#32,'"','´','`',';','.','!',',','?']) then
       in_word := true
      else
       if in_word then
       begin
        Inc(Result);
        in_word := false;
       end;
    end;
   end;
   if in_word then Inc(Result);
  end;
end;
```

Listing 4

```
function Str2Blob(const AString: PChar; const Blob: PBlob): PBlob; cdecl;
{
DECLARE EXTERNAL FUNCTION STR2BLOB
CSTRING(32765), BLOB
RETURNS PARAMETER 2
ENTRY_POINT 'STR2BLOB' MODULE_NAME 'MeineUDFs';
}
begin
 Result := Blob;
 if Assigned(Blob) and Assigned(Blob^.BlobHandle) and Assigned(Blob^.BlobPut
                                                              Segment) then
    Blob^.BlobPutSegment(Blob^.BlobHandle, AString, StrLen(AString));
end;
```

Die Funktion *BLOB_WORD_ COUNT* in Listing 3 demonstriert die Technik des sukzessiven Lesens aus einem Blob-Parameter. Man übergibt einen ausreichend großen Puffer an *BlobGetSegment*, damit ein *MaxSegLen-Block* aufgenommen werden kann. Die Callback- Funktion holt dann ein Scheibchen des BLOBs und gibt die erhaltene Länge an. Beim nächsten Aufruf wird an der erreichten Stelle weitergelesen. Das Auslesen ist nur einmal möglich; ein Rücksetzen oder Positionieren ist nicht vorgesehen. Testen Sie diese Funktion durch einen Aufruf dieser Art: *SELECT BLOB_WORD_ COUNT(blobfeld) FROM tabelle*.

Das Schreiben in ein BLOB-Feld geht ähnlich vonstatten. Allerdings sollte man einen BLOB-Parameter nur beschreiben, wenn er als Funktionsergebnis deklariert ist. Da eine UDF aber eine *TBlob* Struktur nicht selbst initialisieren kann, ist sie hierbei auf die Vorarbeit des Aufrufers angewiesen. Bei InterBase hat man häufig das Problem, einen Textstring mit INSERT in ein BLOB-Feld schreiben zu müssen, was aber mit purem SQL nicht unterstützt wird. Man muss eine UDF verwenden, um den String beim INSERT in einen Text-BLOB zu verwandeln. Listing 4 zeigt eine Lösung, mit der ein Aufruf in der Art *INSERT INTO tabelle (blobfeld) VALUES (STR2BLOB('Text'))* möglich ist.

Eine Besonderheit bei dieser Funktion ist die Deklaration des Funktionsergebnisses. Wie schon gesagt, braucht die externe Funktion einen gültigen *TBlob* Record, um die Daten speichern zu können. Die RETURNS-Klausel der Deklaration bietet eine besondere Option an, mit der man den Rückgabeparameter in der Liste der Eingabeparameter (am Ende) aufführen und dann mit *RETURNS PARAMETER n* identifizieren kann. Die Aufrufsyntax der UDF spart diesen speziellen Eingabeparameter dann aus, aber die Datenbank stellt die erforderliche Struktur an dieser Stelle zur Verfügung.

Natürlich können BLOB-Felder auch ganz andere Daten als Texte enthalten. Häufig werden sie für Bilddaten verwendet. Für solche Daten wären die Beispielfunktionen natürlich nicht anwendbar.

Andere Welten

InterBase ist eine Multi-Plattform-Datenbank. Die aktuelle Version steht außer für Windows auch für Linux und für Solaris zur Verfügung. Im Laufe der Geschichte gab es schon etwa ein Dutzend weiterer Portierungen für andere Betriebssysteme. Die Keimzelle war aber InterBase für VMS auf einer VAX von Digital und auf diesen Ursprung weisen nach wie vor etliche Indizien hin, allen voran die Namenskonvention der Systemtabellen und -felder. Deren Bezeichner werden immer mit *RDB$*.. eingeleitet und das ist absolut VAXischer Stil. Die Zeiten ändern sich, Digital ist vor Jahren von der Firma Compaq übernommen worden, die wiederum gerade im Begriff ist, in Hewlett-Packard aufzugehen. VMS – noch heute technologisch führend – lebt zwar weiter, ist aber als Betriebssystem längst nicht mehr so „hip“ wie einst und verspricht Borland offenbar keinen hinreichenden Marktanteil mehr, weshalb man die InterBase-Variante für Vaters Betriebssystem seit der Version IB4 auf Eis gelegt hat.

Eine gute Eigenschaft, mit deren Hilfe sich UDFs beachtlich verbessern ließen, hat sich das heutige InterBase aus dem Erbgut der VMS-Versionen gerettet. InterBase unterstützt so genannte Parameter-Deskriptoren. Das sind Datenstrukturen, die den Wert eines Parameters zusammen mit Typ- und Zustandsinformationen speichern und es dem Programmierer erlauben, flexibel mit Variablen unterschiedlichen Typs umzugehen. Im Kern handelt es sich quasi um eine frühe Form der heutigen Variants. Im Klartext: Parameter-Deskriptoren würden es uns erlauben, einer UDF einen Parameter zu übergeben, ohne dessen Typ vorher fest zu definieren. Der Parametertyp könnte innerhalb der UDF erkannt und – je nach Anwendungsfall – angemessen behandelt werden. Zudem wäre es auch möglich, den NULL-Zustand eines UDF-Parameters in der externen Funktion zu erkennen. Denn leider werden NULL-Parameter vor dem UDF-Aufruf je nach deklariertem Datentyp einfach in einen 0-Wert oder einen leeren String umgewandelt, sodass die Funktion keine Chance hat, zwischen 0 und NULL zu unterscheiden.

Oberflächlichkeiten

Leider werden diese inneren Werte durch eine in dieser Hinsicht leidenschaftslos implementierte SQL-Schicht verdeckt. Das DECLARE EXTERNAL FUNCTION Statement von InterBase erlaubt nur die Angabe konkreter Parametertypen wie INTEGER oder CSTRING(n), aber keinen Parametertyp DESCRIPTOR oder Ähnliches. Im Firebird-Projekt gibt es inzwischen einen solchen Lösungsansatz, der allerdings als noch nicht völlig ausgetestet bezeichnet wird und nur ansatzweise dokumentiert ist [2, 3]. Es wäre aber doch schön, einen Weg zu finden, die Deskriptoren trotz all dieser Hindernisse einzusetzen.

Wenn Sie eine UDF in einer Datenbank deklarieren, dann werden dadurch – wie für Metadaten üblich – effektiv nur einige Einträge in Systemtabellen erzeugt. Die ein-

Listing 5

```
SELECT * FROM RDB$FUNCTION_ARGUMENTS WHERE RDB$FUNCTION_NAME='TEST_NULL' ORDER BY RDB$ARGUMENT_
                                                                                    POSITION;

/* Ergebnis:
RDB$FUNCTION_NAME       RDB$ARGUMENT_POSITION       RDB$MECHANISM       RDB$FIELD_TYPE...
================        =====================       =============       ==============
TEST_NULL               0                           0                   8
TEST_NULL               1                           1                   8
TEST_NULL               2                           1                   8
TEST_NULL               3                           1                   8
*/
```

schlägigen Systemtabellen sind in diesem Fall *RDB$FUNCTIONS* (definiert die Funktion) und *RDB$FUNCTION_ARGUMENTS* (definiert die Ein- und Ausgabeparameter). Diese sind in den InterBase-Handbüchern beschrieben [4], doch für unsere Zwecke weist diese Dokumentation einige Lücken auf, die wir nur anhand des Open Source [5] schließen können.

Listing 6

```
(* Original Deskriptor-Struktur aus dsc.h:
  typedef struct dsc {
    UCHAR dsc_dtype;
    SCHAR dsc_scale;
    USHORT dsc_length;
    SSHORT dsc_sub_type;
    USHORT dsc_flags;
    UCHAR *dsc_address;
  } DSC; *)

{ alle Konstanten stammen ebenfalls aus dsc.h, siehe
                                    dort fuer weitere }
const
 DSC_null=1;
 DSC_no_subtype=2; //dsc has no sub type specified
 DSC_nullable=4;

const
 dtype_null=0;
 dtype_text=1;
 dtype_cstring=2;
 dtype_varying=3;
 dtype_packed=6;
 dtype_byte=7;
 dtype_short=8;
 dtype_long=9;
 dtype_quad=10;
 dtype_real=11;
 dtype_double=12;
 dtype_d_float=13;
 dtype_sql_date=14;
 dtype_sql_time=15;
 dtype_timestamp=16;
 dtype_blob=17;
 dtype_array=18;
 dtype_int64=19;

type
 PDSC=^TDSC;
 TDSC=packed record
  dsc_dtype,
  dsc_scale: byte;
  dsc_length,
  dsc_sub_type,
  dsc_flags: word;
  dsc_address: pointer;
 end;
```

Innenansichten einer Datenbank

Die entscheidenden Hinweise erhält man tatsächlich aus dem InterBase-Quellcode. Der Aufruf externer Funktionen geschieht im Modul *fun.c* innerhalb der Funktion *FUN_evaluate()*. Dieser internen Funktion werden drei Parameter übergeben, nämlich *FUN function* (Beschreibung der Funktion entsprechend den Daten in den o.g. Systemtabellen), ferner *NOD node* (Liste der Eingangsparameter) und *VLU value* (für den erwarteten Ergebniswert). Wichtig in diesem Zusammenhang sind besonders auch die Quelldateien *val.h* (Struktur *FUN*), *exe.h* (*NOD* und *VLU*), *dsc.h* (DSC) sowie *evl.c* (Funktion *EVL_expr*). Alle diese Dateien befinden sich im Quellcodebaum im Unterverzeichnis *\jrd*.

Wie schon gesagt, lässt sich die Verwendung von Parameter-Deskriptoren beim Deklarieren einer externen Funktion mit SQL nicht definieren. Es ist allerdings möglich, dieses Verfahren durch direktes Schreiben in die Systemtabelle *RDB$FUNCTION_ARGUMENTS* zu aktivieren. Den Übergabemechnismus legt das Feld *RDB$MECHANISM* für jeden Parameter fest.

Wenn wir zum Beispiel eine externe Funktion zum Testen auf den NULL-Zustand eines Parameters kreieren wollen, dann könnten wir diese wie folgt deklarieren:

```
DECLARE EXTERNAL FUNCTION TEST_NULL
INTEGER, INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'TEST_NULL' MODULE_NAME 'MeineUDFs';
```

Die Funktion erwartet drei Integer-Parameter. Der erste sei der zu prüfende Parameter, der zweite der (Integer-)Wert, den die Funktion liefern soll, falls der Kandidat NULL ist, und der dritte der Ergebniswert für den Fall, dass der Kandidat NOT NULL ist. In *RDB$FUNCTION_ARGUMENTS* sind diese Parameter wie in Listing 5 ersichtlich beschrieben. (Einige Felder wurden hier zur Wahrung der Übersichtlichkeit weggelassen.)

Das Funktionsergebnis wird immer in der Position 0 beschrieben. Anschließend folgen die Eingabeparameter von links nach rechts. Im Einklang mit der InterBase Language Reference steht in *RDB$MECHANISM* für *by value* eine *0* und für *by reference* eine *1*. Der *RDB$FIELD_TYPE* ist für alle Parameter *8*, was gemäß der selben Dokumentation INTEGER bedeutet. Aus den Einträgen der Systemtabellen lässt sich tatsächlich der komplette Metacode einer Datenbank zurückentwickeln, was übrigens auch genau der Vorgehensweise von IBConsole und anderen Managementprogrammen beim Extrahieren eines Generierungsscripts aus einer Datenbank entspricht.

Das Feld *RDB$MECHNISM* ist im Handbuch aber nur unvollständig beschrieben; nur die Werte *0* und *1* werden erklärt. Die volle Wahrheit erfährt man erst, wenn man im Quellcode den Aufzählungstyp *FUN_T* (in *val.h*) gefunden hat, dessen mögliche Werte mit diesem Feld korrespondieren. Dieser sieht so aus:

```
typedef ENUM {
  FUN_value,
  FUN_reference,
  FUN_descriptor,
  FUN_blob_struct,
  FUN_scalar_array
} FUN_T;
```

Die ersten beiden Einträge *FUN_value* und *FUN_reference* mit der Wertigkeit *0* bzw. *1* entsprechen genau der Dokumentation. Auch den Wert *3* (für *FUN_blob_struct*) wird man bei BLOB-UDFs tatsächlich vorfinden. Der Wert *4* (*FUN_scalar_array*) kommt in der Praxis nicht vor, da UDFs die ohnehin exotischen Array-Datentypen nicht unterstützen. Ins Auge springt natürlich der Bezeichner *FUN_descriptor*, der die Wertigkeit *2* hat.

Bei InterBase hindert uns nichts daran, auch direkt in die Systemtabellen zu schreiben, ohne dafür die DDL-Befehle zu bemühen. Von dieser Freiheit sollte man allerdings wirklich nur dann Gebrauch machen, wenn man die Konsequenzen dieses Treibens abschätzen kann, denn willkürliches Herumstochern in diesen Tabellen führt unweigerlich ins Chaos. Mit dem folgenden Befehl ändern wir den Mechanismus für den ersten Eingabeparameter auf *FUN_descriptor* ab:

```
UPDATE RDB$FUNCTION_ARGUMENTS SET RDB$MECHANISM
=2 WHERE RDB$FUNCTION_NAME='TEST_NULL' AND
RDB$ARGUMENT_POSITION=1
```

Mit diesem „Hack“ haben wir den Übergabemechanismus des ersten Eingabeparameters also von der gewöhnlichen *by-reference*- auf die undokumentierte *by-descriptor*-Form umgestellt. Nach diesem Eingriff kann ein Trennen aller Verbindungen zu der Datenbank erforderlich sein, damit der interne Metadaten-Cache neu aufgebaut wird und die Änderung wirksam wird. Was nun natürlich noch dringend fehlt, ist eine Implementierung einer Funktion, die mit dieser Art der Parameterübergabe zurechtkommt.

Die Deskriptorstruktur heißt im InterBase-Quellcode DSC und ist in der Datei *dsc.h* aufzufinden. Listing 6 zeigt diese Struktur im Original und – zusammen mit einigen der erforderlichen Konstanten – als Übersetzung in Object Pascal. Sämtliche Konstanten für die Dekodierung der einzelnen Felder sind ebenfalls in *dsc.h* definiert. Den Inhalt dieses Listings fügen Sie einfach in die Unit *udf_def.pas* ein.

Um die Aufgabe der Funktion *TEST_NULL*, nämlich das Testen des ersten (nun per Deskriptor übergebenen) Parameters auf den Zustand NULL, zu implementieren, brauchen wir nur das Feld *dsc_flags* auszuwerten. Ist hier das mit *DSC_null* bestimmte Bit gesetzt, hat der Parameter den NULL-Status und pflichtgemäß wird der Wert im Parameter *2* (anderenfalls *3*) zurückgegeben. Die Implementierung dieser UDF sehen Sie in Listing 7.

Listing 7

```
function Test_Null(const Test: PDSC; var A, B: Integer):
                                              Integer; cdecl;
{
DECLARE EXTERNAL FUNCTION TEST_NULL
INTEGER, INTEGER, INTEGER
RETURNS INTEGER BY VALUE
ENTRY_POINT 'TEST_NULL' MODULE_NAME 'MeineUDFs';
COMMIT;
UPDATE RDB$FUNCTION_ARGUMENTS SET RDB$-
                                              MECHANISM = 2
WHERE RDB$FUNCTION_NAME='TEST_NULL' AND
                                  RDB$ARGUMENT_POSITION=1;
}
begin
 if (Test^.dsc_flags and DSC_null) = DSC_null then
  Result := A
 else
  Result := B;
end;
```

Listing 8

```
SELECT RDB$DESCRIPTION, TEST_NULL(RDB$DESCRIPTION, 1, -1), RDB$RELATION_ID, TEST_NULL(RDB$RELATION_ID, 1,
                                                                                    -1) FROM RDB$DATABASE

/* Ergebnis:
RDB$DESCRIPTION              TEST_NULL             RDB$RELATION_ID              TEST_NULL
==============               =========             ===============              =========
<null>                       1                     130                          -1
*/
```

Um die Funktion zu testen, kann man ihr – wie in Listing 8 gezeigt – beliebige Felder übergeben und zum Beispiel für einen NULL-Zustand eine *1* und für NOT NULL eine *-1* ausgeben lassen, um den Zustand zu unterscheiden. Man sieht, dass das Deskriptor-Verfahren im Gegensatz zur gewöhnlichen UDF-Programmierung tatsächlich die Möglichkeit bietet, den NULL-Zustand innerhalb der externen Funktion zu erkennen. Außer dem dafür benötigten Feld *dsc_flags* hält ein Deskriptor natürlich noch mehr Informationen parat. Wenn man sich die Datentypen der in Listing 8 auf NULL getesteten Felder ansieht, fällt auf, dass diese unterschiedliche Feldtypen haben. Anders als beim Aufruf einer konventionellen UDF werden die übergebenen Parameter nicht vor dem Eintritt in die externe Funktion auf den deklarierten Argumenttyp angepasst, sondern der Datentyp des Parameters wird im Deskriptor mit übergeben. Der Argumenttyp ist sozusagen variabel und verbirgt sich in den Deskriptorfeldern *dsc_dtype*, *dsc_length*, *dsc_scale* und *dsc_sub_type*. Die eigentlichen Argumentdaten stehen an der Adresse, die in *dsc_address* angegeben wird.

Sicherlich sind bezüglich der Deskriptoren noch etliche Fragen offen geblieben, insbesondere die vollständige Auswertung der Argumenttypen und -daten oder auch eine Möglichkeit, mit einer UDF einen Deskriptor als Funktionsergebnis zurückzugeben. Die Klärung dieser und anderer Fragen hat sich die Deutsche InterBase Entwicklerinitiative [6] auf die Fahnen geschrieben.

Diese Art der Erforschung des Verhaltens der Datenbank ist für die meisten interessierten Entwickler, die sich in der Regel nicht zu Open Source Autoren aufschwingen, wohl der Hauptnutzen, den sie aus der Einsicht in den Quellcode gewinnen können.

Ich möchte ganz deutlich darauf hinweisen, dass der oben beschriebene Umgang mit den eigentlich undokumentierten Deskriptoren experimentellen Charakter hat. Auch wenn dieses Verfahren auf den ersten Blick gut zu funktionieren scheint; ausgiebige Tests hinsichtlich der Stabilität habe ich hierzu noch nicht gemacht. Umso mehr liegt es in der eigenen Verantwortung von Entwicklern, die dieses nützliche Merkmal anwenden wollen, selbst die Eignung für den Dauer- und Mehrbenutzerbetrieb sicherzustellen.

Fazit

Nachdem Sie die Verarbeitung aller InterBase-Datentypen in benutzerdefinierten Funktionen kennen gelernt haben, steht Ihnen eine mächtige Möglichkeit zur Verfügung, den Sprachschatz Ihrer Datenbank zu erweitern und an Ihre Anforderungen anzupassen. UDFs sind oft ein nützliches und manchmal auch das einzige Mittel, Verarbeitungsschritte im Server durchzuführen, die man sonst an einen Client delegieren müsste.

Links & Literatur

[1] www.delphi-jedi.org
[2] http://firebird.sf.net
[3] www.cvalde.com/document/using_descriptors_with_udfs.htm
[4] Interbase 6 Language Reference, Kapitel „System Tables and Views”
[5] Karsten Strobel, *Der Entwickler* 6/2001, Den InterBase Open Source selbst compilieren
[6] www.interbase2000.de