# SQL-Akrobatik

## Programmierung benutzerdefinierter Funktionen für InterBase – Teil 1

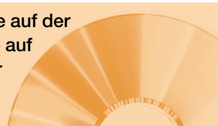
von Karsten Strobel

**Tolle Sache so ein Fertighaus! Die Komponenten werden in großen Teilen angeliefert, und weil es ja ein Serienprodukt und eben keine Maßanfertigung ist, kostet es auch bei weitem nicht so viel. Auf meinem Grundstück ist dann alles in kürzester Zeit installiert. Schade, dass sich die Haustür nicht ganz öffnen lässt, denn sie stößt gegen die Garage des Nachbarn. Das ging aber nicht anders, denn sonst hätte die Terrasse quer über die Bundesstraße verlegt werden müssen. Kein Problem, sage ich mir, dann lege ich halt selbst Hand an. Schon setze ich den Meißel an, um eine neue Türöffnung zu schaffen, da regt sich ein plötzlicher Zweifel. Nicht, dass ich eine tragende Mauer einreiße ...?**

Anwender wollen manchmal schon ulkige Sachen mit ihren Daten anstellen. Mal soll für ein Datum die Nummer der Kalenderwoche, in der es liegt, errechnet werden. Ein weiteres Mal möchte man eine Zahl immer auf volle 25er-Schritte gerundet haben. Bei anderer Gelegenheit wiederum soll ein Kfz-Kennzeichen gemäß einer geheimnisvollen und angeblich amtlichen Vorschrift nach gewissen Regeln immer auf elf Zeichen formatiert werden. Weitere Forderungen, wie das rechtsbündige Ausrichten eines Strings oder eine Modulo-Operation mit zwei ganzzahligen Operanden, nehmen sich dagegen zwar schlichter aus, sie haben mit den vorgenannten, exotischeren Wünschen aber eine Eigenschaft gemeinsam: Sie lassen sich mit purem SQL nur schwer oder gar nicht realisieren. Der SQL-Sprachschatz von InterBase ist mit skalaren Funktionen zur String-Manipulation oder zur Ausführung numerischer Berechnungen, die über die Grundrechenarten hinausgehen, kaum gesegnet. Wenn man die zuvor erwähnten Aufgaben lösen möchte, kann man dies trotz der so nützlichen Stored Procedure-Sprache von InterBase nicht ohne weiteres innerhalb der Datenbank tun, denn es fehlt ganz einfach das nötige Handwerkszeug.

**Quellcode**

Den Quellcode finden Sie auf der aktuellen Profi-CD sowie auf unserer Homepage unter www.derentwickler.de



Auf der Clientseite ist dies kaum ein Problem, denn hier steht ja meist C oder Pascal zur Verfügung, jedoch gilt das nicht immer. Verwendet man zum Beispiel eine Reportingsoftware oder eine Officeanwendung, muss man auf den Hochsprachenkomfort vielleicht auch clientseitig verzichten. Und überhaupt ist es bei einer Client/Server-Lösung guter Stil, möglichst viel Vorverarbeitung auf dem Server stattfinden zu lassen und dem Client nur die Anwenderoberfläche zu überlassen. Genügend Argumente, um auf die so genannten Benutzerdefinierten Funktionen (User Defined Functions, kurz UDF, oder auch External Functions) zurückzugreifen. Damit lassen sich über selbst erstellte ladbare Bibliotheken (bei Windows *.dll*, bei Unix *.so*) beliebige Funktionen in den Sprachschatz von InterBase hineinschmuggeln. Ich werde mich hier auf die populäre Plattform Windows und Delphi beschränken, aber in einem Folgeteil des Artikels auch auf die Unterschiede bei Linux und Kylix eingehen. Grundsätzlich lassen sich UDFs aber natürlich auch mit dem C++-Builder und mit quasi jedem anderen Compiler erstellen, der Funktionsbibliotheken zu erzeugen vermag.

### Von der Stange

Bevor wir aber die Ärmel hochkrempeln und unsere eigenen Funktionen programmieren, sollten wir natürlich sicherstellen, dass wir das Rad nicht neu erfinden. Im Lieferumfang von InterBase befindet sich schon eine fertige UDF-Bibliothek mit rund drei Dutzend Funktionen hauptsächlich mathematischer Natur, darunter aber auch ein paar für String- und logische Operationen. Diese heißt *ib_udf.dll* und befindet sich bei IB6 im Verzeichnis *<InterBase home>\udf*, bei IB5 im Verzeichnis *<InterBase home>\lib*. Das ist auch der Ort, an den alle UDF-Bibliotheken kopiert werden müssen, damit InterBase sie lokalisieren kann. Eine Übersicht der vorgefertigten Funktionen findet man im Handbuch *Language Reference*. Eine weitere nützliche und kostenlose UDF-Sammlung ist die FreeUDF Library von Greg Deatz [1].

Um eine UDF tatsächlich verwenden zu können, genügt es nicht, die DLL-Datei in das richtige Verzeichnis zu kopieren. Man muss die Funktion auch noch deklarieren, um dem Server den formalen Aufbau der Funktion, also Anzahl und Typ

der Parameter und des Ergebniswertes, mitzuteilen. Dies geschieht allerdings individuell für eine Anwenderdatenbank, also nicht global für einen Server. Die deklarierten Funktionen werden damit Teil der Metadaten und ihr Aufbau wird in den Systemtabellen einer Datenbank beschrieben. Wenn der Server mehrere Datenbanken bedient, dann muss man die UDF-Deklaration für jede dieser Datenbanken, in denen man die Funktionen verwenden will, ausführen. Das Syntaxschema des entsprechenden Befehls *DECLARE EXTERNAL FUNCTION* ist wie folgt:

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING
                    (int)[, datatype | CSTRING (int) ...]]
RETURNS {datatype [BY VALUE] | CSTRING (int) | PARAMETER
                                            n} [FREE_IT]
ENTRY_POINT 'entryname'
MODULE_NAME 'modulename';
```

Die konkreten Deklarationsbefehle für die Funktionen der *ib_udf.dll* findet man in der Datei *ib_udf.sql* im oder unterhalb vom Verzeichnis *<InterBase home>\examples*. Dieses Script enthält die Deklarationen aller mitgelieferten Funktionen. Sie können diese Anweisungen in Ihr eigenes Datenbankscript übernehmen bzw. auf eine existierende Datenbank anwenden. Ob Sie dabei nur die wirklich benötigten Funktionen anmelden oder einfach alle deklarieren, ist reine Geschmackssache; nichtverwendete, aber deklarierte UDFs sind kein schwerwiegender Ballast. Aus *ib_udf.sql* hier als Beispiel die Deklaration der *Modulo*-Funktion:

```
DECLARE EXTERNAL FUNCTION mod
INTEGER, INTEGER
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'IB_UDF_mod' MODULE_NAME 'ib_udf';
```

Die Anweisung macht Ihre Datenbank mit der Funktion *mod* bekannt. Der Bezeichner *mod* könnte bei der Deklaration auch anders gewählt werden, ohne dass dazu die DLL geändert werden müsste. Beim Aufrufen von *mod* werden zwei Integer-Parameter übergeben. Das Funktionsergebnis ist vom Fließkommadatentyp *Double Precision*. Der Zusatz *BY VALUE* gibt an, dass die CPU-Register, die nach Rückkehr aus dem Funktionsaufruf das Ergebnis enthalten, nicht als Zeiger auf einen Ergebnisspeicher, sondern unmittelbar als Ergebniswert interpretiert werden sollen. Hinter *ENTRY_POINT* wird der Name der Funktion in der Exporttabelle der DLL und hinter *MODULE_NAME* der Dateiname der Bibliothek angegeben. Der Dateiname sollte ohne Namenserweiterung (also ohne '.dll') geschrieben werden; je nach Plattform fügt InterBase automatisch die richtige Endung an.

Führen Sie die Deklarationsanweisung von *mod* einfach per *isql* aus und testen Sie die Funktion wie folgt:

```
SELECT mod(17, 4) FROM RDB$DATABASE
```

Das Ergebnis sollte 1 sein. Möglicherweise irritiert Sie der Bezug auf *RDB$DATABASE*. Der Inhalt dieser Systemtabelle, von der wir kein einziges Feld verwenden, ist nicht weiter interessant. Aber sie hat eine zuverlässige Eigenschaft, die man bei solchen Tests und ähnlichen Gelegenheiten nutzen kann. Sie enthält nämlich (ähnlich wie bei Oracle die Pseudo-Tabelle *DUAL*) immer genau einen Datensatz. Das garantiert in unserem Beispiel, dass die Funktion *mod(17, 4)* genau ein Mal ausgeführt wird und wir eine Ergebniszeile angezeigt bekommen. Sie können aber einen UDF-Aufruf nicht nur in einem Select-Statement, sondern auch bei Insert, Update und Delete sowie in Stored Procedures und Triggers verwenden.

## Mogelpackung

Vielleicht haben Sie sich gefragt, warum das Funktionsergebnis von *mod* als Fließkommatyp statt als Integer deklariert ist. Das Ergebnis einer *Modulo*-Operation ist ja eigentlich immer eine ganze Zahl, also ist es nicht sonderlich konsequent, hier ein aufwendigeres *Double Precision* zu verwenden, auch wenn der Wert anschließend durchaus einem Integer-Feld zugewiesen und dabei automatisch umgewandelt werden kann. Ich vermute, dass der Autor der *ib_udf*-Bibliothek sich damit um ein lästiges Problem, nämlich die Behandlung einer möglichen Division durch null, herummogeln wollte. Der Versuch

```
SELECT mod(17, 0) FROM RDB$DATABASE
```

liefert nämlich weder eine 0 (das wäre auch nicht korrekt) noch eine Fehlermeldung, sondern den Status INF[INITE], also „unendlich" – ein Status, der in einer Fließkommavariablen durch ein bestimmtes Statusbit angezeigt werden kann. Ein entsprechender Statuscode existiert bei Integer-Variablen bekanntlich nicht. Das ist zwar trickreich, aber nicht sonderlich hilfreich, denn sobald Sie versuchen, ein derart degeneriertes Ergebnis als Integer-Feld zu verwenden, setzt sich InterBase heftig zur Wehr. Auf den Befehl

```
SELECT CAST(mod(17, 0) AS INTEGER) FROM RDB$DATABASE
```

erhält man eine „Arithmetic overflow ..."-Fehlermeldung. Das Unangenehme an diesem Verhalten ist, dass es sich innerhalb von SQL-Anweisungen nur schwer behandeln lässt. Eine gute Alternative wäre, ein Integer-Ergebnis zurückzugeben und den Fehlerfall mit -1 anzuzeigen, da eine erfolgreiche Operation kein negatives Ergebnis haben kann. Dies soll auch unser erstes Beispiel für eine eigene, mit Delphi geschriebene UDF sein.

Kompilieren Sie den Beispielcode in Listing 1 und kopieren Sie die Datei *MeineUDFs.dll* nach *<InterBase home>\udf* (bzw. *\Lib* bei IB5). Alle Quellcodes finden

### Listing 1: Eigene Modulo-Funktion

```
library MeineUDFs;

uses Sysutils;

function Modulo(var A, B: Integer): Integer; cdecl;
{ Deklaration in InterBase:

 DECLARE EXTERNAL FUNCTION MODULO
 INTEGER, INTEGER
 RETURNS INTEGER BY VALUE
 ENTRY_POINT 'MODULO' MODULE_NAME 'MeineUDFs';   }
begin
 if B = 0 then Result := -1 else Result := A mod B;
end;

exports Modulo name 'MODULO';

begin
 IsMultiThread := true;
end.
```

Sie auf der Profi-CD und im Web. Die Quelldatei *MeineUDFs.dpr* enthält die Funktion *Modulo*, einen Einzeiler, der das soeben Beschriebene tut und ein Integer-Ergebnis liefert. Die Deklarationsanweisung ist im Pascal-Quellcode als Kommentar enthalten. Natürlich kann diese Funktion nicht ebenfalls als *mod* deklariert werden, deshalb verwenden wir hier *modulo* als Funktionsnamen. Machen Sie mit dieser neuen Funktion die gleichen Tests wie zuvor. Das Ergebnis der folgenden Abfragen sollte 1 und -1 sein:

```
SELECT modulo(17, 4) FROM RDB$DATABASE;
SELECT modulo(17, 0) FROM RDB$DATABASE;
```

Der Quellcode unserer DLL enthält einige Besonderheiten, die man beim Implementieren von UDFs unbedingt beachten muss: Die Direktive *cdecl* legt die zutreffende Aufrufkonvention (die Vorschrift zur Parameterübergabe) fest und muss für jede Funktion angegeben werden. Mit der *exports*-Klausel werden die sichtbaren Funktionen der DLL und deren Exportnamen festgelegt.

Besonders wichtig ist auch die Anweisung *IsMultiThread := true;* im Initialisierungsteil der DLL. InterBase verwendet intern eine Vielzahl parallel ablaufender Threads, die u.a. für die parallele Verarbeitung von Datenbankabfragen zuständig sind. Wenn zwei gleichzeitig mit der Datenbank verbundene Benutzer jeweils einen SQL-Befehl ausführen, mit dem eine UDF aufgerufen wird, dann können durchaus zu einem Zeitpunkt mehrere Threads Aufrufe in der UDF-Bibliothek ausführen. Also muss die ganze Bibliothek unbedingt Thread-sicher programmiert werden. Das dynamische Speichermanagement des Delphi-Laufzeitsystems verhält sich aber nur dann thread-safe, wenn die globale Variable *IsMultiThread* auf *true* gesetzt wird. Sonst wird auf die Gefahren des Wiedereintritts in die Speicherallokierungs-Funktionen zugunsten der Performance keine Rücksicht genommen. Schon die Verwendung eines Pascal-Strings setzt die Mühlen der dynamischen Speicherverwaltung in Gang. Darum muss dieser Schutz unbedingt eingeschaltet werden, weil es sonst zu sehr tückischen Fehlern kommen kann, die sich direkt auf den Datenbankprozess auswirken können, da die DLL in demselben Adressraum operiert und somit die Freiheit besitzt, die internen Datenstrukturen von InterBase zu überschreiben. Jede mit Delphi geschriebene UDF-Bibliothek, die diese Anweisung nicht enthält, muss als gefährlich gelten.

Bei unserer *Modulo*-Funktion sind die Eingabeparameter A und B als „var-Parameter“ deklariert, werden also per Referenz und nicht als Wert übergeben. Parameter einer UDF werden immer auf diese Art – als Zeiger auf eine von InterBase angelegte Variable, die den eigentlichen Wert enthält – bereitgestellt. Dem muss man beim Deklarieren der UDF auf der Delphi-Seite Rechnung tragen, indem man den passenden Parameter-Mechanismus wählt. Es ist sehr wichtig, dass man in der Programmierung der UDF und bei der Deklaration auf Seiten der Datenbank absolute Übereinstimmung der Ein- und Ausgabeparameter wahrt. Nichts hindert Sie daran, die Parameter einer Externen Funktion falsch zu deklarieren und diese Funktion auch aufzurufen, denn es gibt keinerlei automatische Überprüfung der Parameterliste. Nur wird diese UDF wahrscheinlich nicht funktionieren und in aller Regel beim Aufruf zu schweren Fehlern bis hin zum Absturz des Serverprozesses führen.

Tabelle 1 zeigt eine Liste der wichtigsten InterBase-Datentypen mit den dazu passenden Parametertypen einer in Delphi geschriebenen UDF. Bei einigen Typen muss man zwischen InterBase V6 mit Dialekt 3 und der V5 bzw. dem dazu weitgehend kompatiblen Dialekt 1 der V6 unterscheiden. Einige der Datentypen werden nicht direkt auf Delphi-Datentypen abgebildet, sondern als Zeiger auf spezielle Strukturen übergeben (Varchar, Datumstypen, Blobs). Mit diesen Strukturen werden wir uns später zum Teil noch auseinander setzen.

| Für UDF deklarierter Datentyp | Delphi-UDF-Eingabeparameter | |
|---|---|---|
| | InterBase 6, Dialekt 3 | InterBase 5, Dialekt 1 |
| SMALLINT | var smallint | |
| INTEGER | var integer | |
| FLOAT | var single | |
| DOUBLE PRECISION | var double | |
| NUMERIC(n,m), n in 1..4 | var smallint | |
| NUMERIC(n,m), n in 5..9 | var integer | |
| NUMERIC(n,m), n in 10..18 | var int64 | var double |
| DECIMAL(n,m) | wie NUMERIC(n,m) | |
| CSTRING(n) | const Pchar | |
| VARCHAR(n) | const Pvarchar | |
| CHAR(n) | const Pchar (ohne terminierende #0) | |
| DATE | var integer | const PISC_QUAD |
| TIME | var cardinal | existiert nicht |
| TIMESTAMP | const PISC_QUAD | existiert nicht |
| BLOB | const Pblob | |

Tabelle 1: Abbildung UDF-Deklarationstypen auf Delphi-Datentypen

### Textaufgaben

Häufig möchte man in UDFs mit Strings hantieren, denn in diesem Bereich sieht es in der SQL-Sprache von InterBase besonders mager aus. Die mitgelieferte Bibliothek *ib_udf.dll* bietet schon *ltrim()*, *rtrim()*, *strlen()* und *substr()* an, womit zwar das Notwendigste abgedeckt ist, aber immer noch viele Wünsche offen bleiben. Eine oft benötigte Funktion ist das rechtsbündige Ausrichten eines Strings auf eine bestimmte Länge mit einem beliebigen Füllzeichen, die wir als nächstes Beispiel realisieren werden.

Die InterBase String-Datentypen VARCHAR(n) und CHAR(n) können zwar als UDF-Parameter deklariert werden, aber sie sind etwas umständlich auszuwerten. VARCHAR wird mit einem 2-Byte-Längenfeld vor dem eigentlichen String übergeben, während CHAR eine fest deklarierte Länge hat und weder mit einem Einleitungs- noch einem Abschlusszeichen ausgestattet ist. Um uns die Sache einfacher zu machen, bietet das *DECLARE EXTERNAL FUNCTION*-Statement alternativ den Typ CSTRING(n) für das Handling von Stringparametern an. Dieser Typ ist eigentlich kein echter Datentyp, denn er kann nicht für ein Datenbankfeld verwendet werden, sondern bewirkt eine Umwandlung in einen nullterminierten String. Die Länge (max. 32767 Byte) wird in *n* angegeben, wobei das automatisch angehängte Nullzeichen mitgezählt wird. Dieser Typ kann auch als Funktionsergebnis deklariert werden.

Im Gegensatz zu numerischen Funktionsergebnissen kann man einen String nicht *BY VALUE* an die Datenbank zurückgeben, sondern muss eine Referenz abliefern, also einen Zeiger auf eine Stelle im Speicher, an der die Stringdaten stehen. Dies ist eine der größten Fehlerquellen bei der UDF-Programmierung, und ich möchte mich im nächsten Beispiel bewusst auf dieses Glatteis begeben und zunächst einen oft gemachten Fehler begehen, um diesen Punkt deutlich herauszuarbeiten.

In Listing 2 ist eine bestechend einfach wirkende Implementierung der *RAlign*-Funktion gezeigt. Die drei Eingangsparameter enthalten den Ursprungsstring, das Auffüllzeichen und die gewünschte Länge des Ergebnisstrings. Die String-Parameter einschließlich Funktionsergebnis sind als CSTRING-Pseudotyp deklariert, können also als Strings mit abschließendem #0-Zeichen behandelt werden. Delphi macht den Umgang mit nullterminierten Strings bekanntlich sehr einfach. Man kann durch einfache Cast-Ausdrücke zwischen Pascal-Strings und nullterminierten Strings quasi hin- und herschalten. So gesehen erscheinen die beiden Anweisungen, mit denen das gewünschte Ergebnis zunächst als Pascal-String zusammengebaut und dann mit einem *PChar()* Cast-Ausdruck auf den Ergebnistyp der Funktion angepasst wird, durchaus geschickt.

Das Tückische an diesem Lösungsansatz ist, dass er – flüchtig betrachtet – durchaus funktioniert. Setzen Sie die neue Funktion in den UDF-Quellcode unter die *Modulo*-Funktion und kompilieren Sie die Bibliothek neu. Sie müssen die neue DLL wieder in das richtige Verzeichnis kopieren und dabei die vorherige Version ersetzen. Falls die alte DLL noch in Verwendung ist, können Sie sie nicht überschreiben und Sie müssen alle offenen Sitzungen zur Datenbank beenden, damit sie von InterBase wieder entladen wird. Wenden Sie das DECLARE-Statement aus dem Kommentarblock im Delphi-Quelltext an und probieren Sie die Funktion aus:

```
SELECT ralign('100', '*', 6) FROM RDB$DATABASE;
```

Als Ergebnis wird `'***100'` ausgegeben, was eigentlich gar nicht so schlecht aussieht.

#### Listing 2: Fehlerhafter Ansatz für RAlign

```
function RAlign(const AStr, AFillChar: PChar; var ALen:
                                   Smallint): PChar; cdecl;
{ Deklaration in InterBase:

  DECLARE EXTERNAL FUNCTION RALIGN
  CSTRING(80), CSTRING(2), SMALLINT
  RETURNS CSTRING(80)
  ENTRY_POINT 'RALIGN' MODULE_NAME 'MeineUDFs';  }
var s: string;
begin
  //Falscher Ansatz...
  s := StringOfChar(AFillChar^, ALen-Integer
                           (StrLen(AStr)))+string(AStr);
  Result := PChar(s);
end;

exports RAlign name 'RALIGN';
```

#### Listing 3: UDFs im Stress-Test

```
CREATE PROCEDURE TESTUDF (AFILL CHAR(1), ACOUNT
INTEGER)
RETURNS (RRESULT VARCHAR(80))
AS
DECLARE VARIABLE I INTEGER;
BEGIN
  I = 1;
  WHILE (I <= ACount) DO
  BEGIN
    SELECT ralign(:I, :AFill, modulo(:I, 50))
                    FROM rdb$database INTO :RRESULT;
    SUSPEND;
    I = I + 1;
  END
END
```

### Nicht stressresistent?

Die Schwäche dieser Implementierung wird erst deutlich, wenn Sie die Funktion unter den Bedingungen eines Mehrbenutzerbetriebs testen und etwas unter Stress setzen – ein Test, der für die Qualitätskontrolle eigener und auch fremder UDFs durchaus sinnvoll ist. Listing 3 zeigt eine Stored Procedure, die *RAlign* in einer Schleife immer wieder aufruft. Rufen Sie diese Prozedur per SELECT aus mehreren parallelen (!) ISQL-Sitzungen gleichzeitig auf, und leiten Sie die Ausgabe in verschiedene Dateien um:

```
SELECT * FROM TESTUDF('*', 1000); /* erzeugt 1000
                                          Ausgaben */
```

Vergleichen Sie die Ausgabedateien miteinander. Sie werden feststellen, dass Sie nicht völlig gleich sind, obwohl sie es eigentlich sein sollten. Wenn Sie die Ausgaben näher untersuchen, stoßen Sie mitunter auf Stellen, die vom regelmäßigen Muster abweichen und etwa so aussehen können:

```
103
*104
**105
***106
****107
*******8
******10
*******1
    ********111
```

Dies ist ein deutliches Zeichen dafür, dass bei der Übernahme des *RAlign*-Funktionsergebnisses etwas furchtbar schief geht. Was ist hier los? Gegen die Verwendung von Pascal-Strings innerhalb einer UDF ist zwar grundsätzlich nichts einzuwenden, aber die Rückgabe eines nullterminierten Strings durch Casting des Pascal-Strings *s* ist eine nachlässige Methode, denn die lokal deklarierte Variable *s* hat eine begrenzte Lebensdauer. Der Delphi-Compiler erzeugt Code, der *s* bei Bedarf im dynamischen Speicher erzeugt und beim Verlassen der Funktion wieder freigibt. Der Inhalt von *s* steht nach der Freigabe dieses Speicherblocks zwar in der Regel noch an derselben Stelle, aber diese Information ist flüchtig, denn bei nächster Gelegenheit könnte der Delphi-Speichermanager diesen schon freigegebenen Block wieder für einen anderen Zweck verwenden, etwa um eine neue dynamische Variable zu erzeugen. Das Verhängnis dieser Reinlichkeit kommt erst ans Tageslicht, wenn mehrere Datenbanksitzungen parallel die UDF aufrufen, womit die Wahrscheinlichkeit stark steigt, dass tatsächlich derselbe Speicherblock von einem Thread neu verwendet wird, während InterBase das Ergebnis eines früheren UDF-Aufrufs im Kontext eines anderen Threads gerade noch auswertet. Unter ungünstigen Umständen kann dies auch zu einer Schutzverletzung oder zu internen Inkonsistenzen des Servers führen.

Um diese Probleme zu vermeiden, bietet InterBase einen Mechanismus an, bei dem die UDF beim Speichermanager der Datenbank einen Ergebnisbuffer allokiert, der nach der Rückkehr aus der Externen Funktion von InterBase wieder freigegeben wird. Um InterBase mitzuteilen, dass nach dieser Methode verfahren werden soll, muss im Deklarations-Statement hinter dem Ergebnisdatentyp das Schlüsselwort *FREE_IT* angegeben werden.

Listing 4 zeigt eine bessere Version der *RAlign*-Funktion. Kommentieren Sie einfach den fehlerhaften Code von dem vorherigen Versuch aus und setzen Sie die neue Version an deren Stelle. In der Datenbank muss die Deklaration richtig gestellt werden, d.h., Sie müssen die alte Deklaration mit *DROP EXTERNAL FUNCTION RALIGN* entfernen und dann die neue Deklaration (mit der *FREE_IT*-Direktive) anwenden. Bei dieser Variante habe ich zugunsten der Performance ganz auf Pascal-Strings verzichtet. Der wesentliche Unterschied ist aber, dass der Ergebnisstring in einem durch *ib_util_malloc()* angeforderten Buffer abgelegt wird, auf den die Funktion einen Pointer zurückgibt. Dieser Buffer wird (wegen *FREE_IT*) von InterBase wieder freigegeben. Ohne *FREE_IT* würde diese Methode ein Speicherleck verursachen. Die Funktion *ib_util_malloc()* ist in der *ib_util.dll* implementiert, die mit InterBase mitgeliefert wird und sich im *\udf-* bzw. *\lib*-Verzeichnis befindet. Wichtig: Bei InterBase V5 müssen Sie die *ib_util.dll* manuell von *\lib* nach *\bin* kopieren, damit die Funktion zur Laufzeit eingebunden werden kann.

### Listing 4: Korrekte Version von RAlignt

```
function ib_util_malloc(l: integer): pointer; cdecl; external 'ib_util.dll';

function RAlign(const AStr, AFillChar: PChar; var ALen: Smallint): PChar; cdecl;
{ Deklaration in InterBase:

 DECLARE EXTERNAL FUNCTION RALIGN
 CSTRING(80), CSTRING(2), SMALLINT
 RETURNS CSTRING(80) FREE_IT
 ENTRY_POINT 'RALIGN' MODULE_NAME 'MeineUDFs';  }
var
 c: Char;
 L1, L2, i: Integer;
begin
 Result := nil;
 if ALen <= 0 then Exit;
 c := AFillChar^;
 if c = #0 then c := #32;
 L1 := StrLen(AStr);
 if ALen > L1 then L2 := ALen else L2 := L1;
 Result := ib_util_malloc(L2+1);
 StrCopy(Result+(L2-L1), AStr);
 for i := 0 to L2-L1-1 do (Result+i)^ := c;
end;
```

### Listing 5: Nachkommastellen als Ganzzahl ausgeben

```
function Frac_92(var Value: Integer): Integer; cdecl;
{ Deklaration in InterBase:

 DECLARE EXTERNAL FUNCTION FRAC_92
 NUMERIC(9,2)
 RETURNS INTEGER BY VALUE
 ENTRY_POINT 'FRAC_92' MODULE_NAME 'MeineUDFs';  }
begin
 Result := Value mod 100;
end;

exports Frac_92 name 'FRAC_92';
```

## Schiebung

Noch ein wichtiges Betätigungsfeld für UDFs sind numerische Operationen. Selbst eine so einfache Aufgabe, wie den Bruchanteil einer reellen Zahl auszugeben, verursacht bei purem SQL schon echte Kopfschmerzen.

Listing 5 zeigt hierfür ein Lösungsbeispiel, bei dem ein Parameter von Typ *NUMERIC(9,2)* an die UDF übergeben wird. Wie schon in Tabelle 1 angegeben, ist auf Delphi-Seite dieser Parameter formal als *„var Value: Integer“* deklariert. InterBase speichert und verarbeitet diesen Datentyp nämlich intern als 32-Bit-Ganzzahl und merkt sich zusätzlich die Position des Dezimalkommas. Konsequent wird der Parameterwert auch als Integer an die UDF übergeben, die die Kommaposition einfach „wissen“ muss. Wenn Sie z.B. den Aufruf

```
SELECT frac_92(1234.56) FROM RDB$DATABASE;
```

testen, dann wird an die Funktion der Integer-Wert 123456 übergeben. Daraus liefert *Result := Value mod 10* wie gewünscht das Ergebnis 56.

## Fazit

Die Benutzerdefinierten Funktionen von InterBase sind ein äußerst flexibles Mittel, mit dem man die SQL-Sprache nach Gusto erweitern kann. Dabei ist die Gefahr dieser Technik nicht zu unterschätzen, und es erfordert genaue Kenntnisse und intensives Planen und Testen, um die katastrophalen Folgen fehlerhaft programmierter UDFs sicher zu vermeiden. Im nächsten Teil dieses Artikels werden wir noch tiefer in diese brisante Materie einsteigen.

**Literatur und Links:**

[1] ftp://ftp.ibphoenix.com/downloads/freeudflib.zip