Johannes Meiners, Wilhelm Nüßer

# SAP-Schnittstellen-programmierung


Galileo Press

# Inhalt

# Einleitung

Dieses Buch behandelt die wichtigsten Möglichkeiten für die Entwicklung von Schnittstellen zwischen R/3-Systemen und externer Software. Den Schwerpunkt bildet dabei die RFC-Bibliothek. Darüber hinaus werden aber auch neuere technologische Ansätze von SAP behandelt: Hier liegt der Fokus auf der aktuellen BAPI-Technologie und dem Java Connector (JCo).

Ein Buch über SAP-Schnittstellenprogrammierung, mit Schwerpunkt auf der RFC-Bibliothek, gar mit C-Programmen, aber kein Wort zu Webservices, SAP XI und .Net? Sicher, »reden« muss man wohl manchmal mit dem SAP-System, Schnittstellen sind also wichtig, aber ist die Technologie der Remote Function Calls (RFC) – entstanden in der alten R/2-Zeit – heute im Zeitalter von J2EE, .Net, Webservices und wie die Technologien alle heißen noch interessant und wichtig? Selbst SAP propagiert doch mit erheblichem Marketing-Aufwand diese modernen Ansätze und will damit aus der Ecke eines Anbieters rein proprietärer Lösungen und Protokolle heraus.

Nun, sicher ist diese Frage rhetorischer Natur, denn wenn wir sie mit einem »Nein« beantwortet hätten, wäre dieses Buch nicht entstanden. Unserer Ansicht nach sprechen einige gute Gründe für die Behandlung dieses Themas im Rahmen eines Buches:

1. Die RFC-Schnittstelle ist ein zentraler Bestandteil des SAP-Applikationsservers. Nach wie vor wird sie SAP-intern und von vielen Kunden für die Realisierung von z.T. sehr komplexen Lösungen verwendet.
2. Ein Großteil der SAP-Kunden setzt immer noch Releases der SAP-Software ein, die nicht oder nur sehr bedingt die oben genannten neuen Technologien unterstützen. Diese Kunden sind weiterhin ausschließlich auf die RFC-Bibliothek angewiesen.
3. Selbst wenn jedoch neue Technologien eingesetzt werden können, basieren sie entweder auf der RFC-Schnittstelle oder/und bieten nur einen Teil der Funktionalität dieser etablierten Schnittstelle. Wir zeigen dies im letzten Kapitel anhand des Java Connector (JCo) auf.
4. Last but not least sind die Programmiermodelle, die bei der RFC-Programmierung eine Rolle spielen, in viele Client-Server-Umgebungen übertragbar. Es gibt also sogar didaktische Aspekte, warum sich ein Entwickler die RFC-Bibliothek einmal genauer anschauen sollte – auch wenn es einfachere APIs als die RFC-Schnittstelle gibt.

Wir haben – um insbesondere diesem letzten Aspekt Rechnung zu tragen – einen recht langsamen, vorsichtigen Einstieg in die Thematik gewählt. Erfahrene SAP-

Entwickler werden an der einen oder anderen Stelle auf Altbekanntes stoßen und vorblättern, ein Einsteiger in die SAP-Welt benötigt aber Hintergrundinformationen, die es ihm ermöglichen, nach der Lektüre des Buches selbstständig voranzuschreiten. Das Buch gliedert sich daher in neun Kapitel:

- Das **Kapitel 1** ist eine kurze Einführung in die Grundbegriffe des R/3-Systems. Hier stellen wir wesentliche Elemente wie den SAP-Applikationsserver, die Workprozesse und das Gateway vor.
- In **Kapitel 2** folgt eine Zusammenstellung der wichtigsten Tools der ABAP Workbench. Diese beiden ersten Kapitel können von erfahrenen SAP-Entwicklern ohne weiteres übersprungen werden.
- Im **Kapitel 3** beginnt dann die eigentliche Programmierung mit den RFCs der SAP. Wir fangen hier mit der Programmiersprache C an, da sie die originäre Sprache der RFC-Schnittstelle ist. Sie lernen in diesem Kapitel einfache Client-Server-Anwendungen kennen.
- Das **Kapitel 4** vertieft die eher didaktisch motivierten Ansätze des Kapitels 3. Datentypen in der RFC-Bibliothek und im R/3-System sind hier ein wichtiges Thema. Dieses und das vorige Kapitel sind das Kernstück des Buches.
- Da Sie nach der Lektüre der ersten Kapitel in der Lage sind, auch durchaus komplexe Programme zu entwerfen und zu realisieren, liegt es nahe, im **Kapitel 5** Werkzeuge zur Fehlerdiagnose und zum Troubleshooting vorzustellen. Auch dieses Kapitel kann ein SAP-Kundiger überschlagen, da die meisten Werkzeuge bekannt sein dürften.
- Das **Kapitel 6** greift dann einige weniger bekannte und durchaus als »fortgeschrittene Programmierung« zu bezeichnende Aspekte der RFC-Schnittstelle auf. Hier behandeln wir z. B. Themen wie tRFCs, qRFCs und Parallelverarbeitung.
- Eine konzeptionell sehr wichtige Form der Schnittstellen des SAP-Systems, die BAPIs (*Business Application Programming Interfaces*), diskutiert das **Kapitel 7**. BAPIs stellen einen von SAP definierten objektorientierten Weg zu stabilen Schnittstellen dar – und werden letztlich über die RFC-Schnittstelle angesprochen.
- Eine der wichtigsten Anwendungen finden die BAPIs im Zusammenhang mit den ActiveX-Controls. Dieses Zusammenspiel ist Inhalt des **Kapitels 8**. Hier verlassen wir den bis dahin weitgehend plattformunabhängigen Rahmen und konzentrieren uns auf die Microsoft-Welt.
- Das **Kapitel 9** geht in dieser Hinsicht den umgekehrten Weg: Der Java Connector (JCo) benötigt als Laufzeitumgebung nur die übliche Java Virtual Machine und ist dann auf allen Plattformen lauffähig. Er stellt für uns auch den Über-

gang in die oben angesprochenen modernen Technologien dar. Er ist z. B. Teil der Java Connector Architecture und damit der Anbindung von SAP an J2EE-Server.

Dabei sind die C/C++-Programmbeispiele in den Kapiteln 3 bis 6 auf Windows-2000-Betriebssystemen entwickelt worden, so dass bei Entwicklungen für andere Plattformen entsprechende Anpassungen vorgenommen werden müssen.

Wir hätten uns im Rahmen dieses Buches noch viele weitere Themen vorstellen können. Gerade die aktuell viel zitierten Webservices liefern technologisch gesehen eine echte Alternative zum RFC. Uns war jedoch an einer Fokussierung auf Themen gelegen, die man als »klassisch« in dem Sinn bezeichnen kann, dass sie für eine große Zahl an laufenden Systemen relevant sind. Webservices sind eine Option für die Zukunft; die zum jetzigen Zeitpunkt noch nicht abgeschlossene Konsolidierung der Standards und der Software ließen das Thema für dieses Buch – noch – nicht passend erscheinen.

Das Buch wendet sich primär an Entwickler, also an die Personen, die versuchen müssen, aus den Ideen des Beraters ein »sauberes« Programm zu gestalten. Wir zeigen ihnen, welche Möglichkeiten zur Gestaltung von Schnittstellen zwischen R/3-System und externen Systemen bestehen. Wir wenden uns aber auch an Berater; sie müssen zwar nicht selbst ihr Konzept für eine Schnittstelle umsetzen, dennoch ist es in der Planungsphase häufig hilfreich, wenn man ein Gefühl dafür hat, welche Alternativen für die Realisierung von Schnittstellen bestehen.

Um Ihnen das leidige Abtippen der zahlreichen langen Beispiele zu ersparen, finden Sie den Quellcode der vollständigen Programme auf der Katalogseite zum Buch unter *www.sap-press.de*. Auf dieser Website finden Sie außerdem ein Forum zum Buch, in dem Sie sich mit anderen Lesern austauschen und mit den Autoren in einen fachlichen Kontakt treten können.

An der Entstehung eines Buches wie dem vorliegenden sind immer viele Menschen direkt und indirekt beteiligt. Johannes Meiners dankt besonders Herrn Prof. Dr. Nowack von der Fachhochschule Münster, University of Applied Sciences, der ihm den Zugang zu den dortigen SAP-Labors ermöglichte. Weiterhin geht sein Dank an die Abteilungen Entwicklung und Technical Consulting der itelligence AG, Bielefeld, für ihre hilfreiche Unterstützung bei der Beantwortung fachlicher Fragen. Wilhelm Nüßer dankt im Besonderen der Heinz Nixdorf Stiftung, ohne deren großzügige Unterstützung ihm die Beteiligung an diesem Buch nicht möglich gewesen wäre, und der Fachhochschule der Wirtschaft (FHDW), Paderborn. Gemeinsam danken wir auch den Mitarbeitern von SAP – allen voran Herrn Thomas Becker –, die uns mit Rat und Tat zur Seite gestanden haben und deren Unterstützung wesentlich zur Fertigstellung des Buches beigetragen hat. Ferner möchten wir uns bei Florian Zimniak vom Verlag Galileo Press für die freundliche Betreuung bedanken.

# 4 Grundlagen der RFC-Programmierung

Im vorangegangenen Kapitel haben wir einfache externe Clients und Server entwickelt, die mit dem R/3-System Daten austauschen. In diesem Kapitel werden nun folgende Aspekte der RFC-API vertieft:

- Type-Mapping
- Umgang mit Datenaggregaten
- Nachrichtenschleife

## 4.1 Type-Mapping und Datenaggregate

Eines der gemeinsten Computergesetze von Murphy lautet:

> *»Programmieren ist wie Romanschreiben. Erst denkt man sich ein paar Typen aus, und dann muss man sehen, wie man mit ihnen zurechtkommt.«*[1]

Damit Sie beim Umgang mit Datentypen in Zukunft weniger Probleme haben, sollen Ihnen die folgenden Abschnitte zeigen, welche Aspekte bei der Übertragung von Datentypen beachtet werden sollten. Behandelt werden dabei u.a.:

- Probleme bei der Übertragung von Strings
- Umgang mit gepackten Zahlen

### 4.1.1 Generische Datentypen

Dieser Abschnitt zeigt, was ein generischer Datentyp überhaupt ist und warum er nicht in einem RFC-fähigen Funktionsbaustein verwendet werden darf. Der Ausgangspunkt für unsere Überlegungen ist die Anforderung, den Namen eines Kunden von einem R/3-System an ein externes Programm zu senden.

Der gestandene ABAP-Entwickler könnte der Versuchung erliegen, in der Schnittstelle des Funktionsbausteins einfach einen zusätzlichen Exporting-Parameter mit dem Datentyp `c` zu deklarieren. Aber Halt! Versuchen Sie im Anschluss an die Definition des Parameters den Funktionsbaustein zu übersetzen, erscheint folgende seltsame Fehlermeldung: *Beim RFC sind keine generischen Typen zugelassen*.

Was wurde falsch gemacht? Schließlich ist es in einem normalen Funktionsbaustein zulässig, eine Variable mit dem Datentyp `c` zu deklarieren. Die Antwort auf dieses Phänomen soll die Betrachtung eines in ABAP geschriebenen Unterprogramms geben.

1 Dieses und die in späteren Kapiteln zitierten Computergesetze sind entnommen aus: Graf, Joachim: *Murphy's gemeinste Computergesetze*. Markt+Technik 1998.

Wenn an einen ABAP-Entwickler die Anforderung gestellt wird, ein Unterprogramm zu entwickeln, das die Länge eines Strings ausgibt, würde er in etwa folgendes Unterprogramm entwerfen:

```
FORM write_string_length USING VALUE(ip_string) TYPE c.
  DATA: lp_size TYPE i.
    lp_size = strlen( ip_string ).
    WRITE: / 'Die Länge des Strings ist: ', lp_size.
ENDFORM.
```

Werden in einem ABAP-Report zwei Variablen des Typs `c` deklariert, kann das obige Unterprogramm ohne weiteres die Länge der Strings ermitteln.

```
REPORT zprintstrlen.
DATA: Vorname(20) TYPE c VALUE 'Luke'
    , Nachname(40) TYPE c VALUE 'Skywalker'.
START-OF-SELECTION.
PERFORM write_string_length USING Vorname.
PERFORM write_string_length USING Nachname.
```

Es ist erstaunlich, dass das Programm einwandfrei funktioniert. Für den Typ `c`, der in der Schnittstelle des Unterprogramms definiert wurde, ist die Standardgröße ein Byte. Er kann somit nur ein Zeichen aufnehmen. Die Variablen `Vorname` und `Nachname` sind aber 20 bzw. 40 Zeichen groß. Dennoch wird die korrekte Länge des Vor- und Nachnamens ausgegeben.

Dies liegt daran, dass die Variable des Typs `c` tatsächlich in der Schnittstelle des Unterprogramms nicht vollständig definiert wurde. Die Größe der Variablen wurde nicht festgelegt. Diese Art von Schnittstellenparametern bezeichnet man auch als *generische Parameter*. Bei einem generischen Schnittstellenparameter ergänzt das SAP-System die fehlenden Eigenschaften zur Laufzeit selber, indem die Eigenschaften vom Datentyp derjenigen Variable übernommen werden, die an das Unterprogramm zur Laufzeit übergeben wird. Der Schnittstellenparameter `IP_STRING` ist somit einmal 20 und einmal 40 Byte groß. Neben dem Typ `c` gehören zu den generischen Schnittstellenparametern auch noch die numerischen Strings, die gepackten Zahlen sowie der Typ `x`.

Genau dasselbe Prinzip zur Ergänzung fehlender Typinformationen findet auch bei den Schnittstellenparametern in Funktionsbausteinen Anwendung. Hier ergibt sich jedoch ein Problem, falls der Funktionsbaustein RFC-fähig ist. Bei einem RFC-Aufruf werden nur Bytes zwischen Sender und Empfänger übertragen, aber keine Informationen über Datentypen. Der Sender oder Empfänger einer Bytefolge muss daher selbst wissen, wie er die Bytefolge zu interpretieren hat. Dies kann er aber nur, wenn er die Anzahl der übertragenen Variablen und ihre Größe kennt. Kann der Empfänger diese Informationen nicht selbst bereitstellen,

besteht die Gefahr, dass die Bytefolge falsch interpretiert wird und Daten falsch verarbeitet werden. Deshalb müssen alle Schnittstellenparameter eines RFC-fähigen Funktionsbausteins voll qualifiziert sein. Voll qualifiziert bedeutet, dass Datentyp und Größe des Schnittstellenparameters eindeutig festgelegt sind.

Bei einem Integer-Parameter gibt es keine Probleme. Der Datentyp `i` besitzt in ABAP die festgelegte Größe von vier Byte. Bei einem echten generischen Datentyp gibt es dagegen folgende Möglichkeiten, ihn vollständig zu qualifizieren:

- Bezugnahme auf ein entsprechendes Tabellen- oder Strukturfeld, das die gewünschten Eigenschaften besitzt. Der Schnittstellenparameter referenziert mit `LIKE` auf das Feld der Tabelle oder Struktur.
- Bezugnahme auf ein Datenelement oder eine Domäne des Data Dictionary. Hier referenziert der Schnittstellenparameter mit dem Schlüsselwort `TYPE` auf das Datenelement oder die Domäne.

### 4.1.2 Die Character-Datentypen

Der vorangegangene Abschnitt zeigte, was bei der Deklaration von Strings in der Schnittstelle eines Funktionsbausteins zu beachten ist. Wir betrachten nun, welche Probleme es bei der Übertragung von Zeichenketten geben kann.

Zu den Character-Datentypen gehören `RFC_CHAR`, `RFC_NUM`, `RFC_DATE` und `RFC_TIME`. Die drei letztgenannten Datentypen unterscheiden sich vom Typ `RFC_CHAR` dahin gehend, dass die Strings nur numerisch sein sollten.

Das Übertragen eines einzelnen Zeichens ist unproblematisch. Client und Server senden bzw. empfangen immer dasselbe Zeichen. Anders verhält es sich bei Zeichenketten. Bei ihnen besteht das Problem primär darin, dass C/C++ und ABAP unterschiedliche Methoden zur Erkennung des Stringendes besitzen.

In C/C++ ist ein String eine Null-terminierte Zeichenkette. Eine Zeichenkette sieht somit in C/C++ intern so aus:

| S | t | r | i | n | g | \0 |
|---|---|---|---|---|---|---|

Die Bedeutung der Null (`\0`) besteht darin, dass sie das Ende der Zeichenkette signalisiert. Fehlt sie, erkennt C/C++ das Ende nicht.

In C/C++ gelten für die Arbeit mit Strings folgende Regeln:

- Strings sind mit einer Null zu terminieren.
- Die Größe des Character-Felds ist die Summe aus der Anzahl der Zeichen plus eins für die abschließende Null. Soll der Vorname 30 Zeichen umfassen, ist das entsprechende Character-Feld 31 Zeichen groß.

- Character-Felder müssen immer typengerecht mit `NULL` initialisiert werden, da im Allgemeinen in C/C++ keine automatische typengerechte Initialisierung stattfindet.

Dies gilt für ABAP jedoch nicht. Dort gelten folgende Regeln:

- Die Größe des Strings wird allein durch die Länge des Wertes determiniert, der in ihm gespeichert werden soll. Eine Variable `Vorname`, die zehn Zeichen umfassen soll, wird in ABAP deklariert als:

```
data: Vorname(10) type c.
```

- Eine Zeichenkette besitzt als Initialwert immer den Wert ' ' – sprich Space. Dieses Zeichen entspricht dem Wert $20_{hex}$ in der ASCII-Tabelle.

Diese Regeln lassen folgende Schlüsse zu:

- In ABAP und C/C++ werden Strings, die Werte gleicher Größe speichern sollen, eine unterschiedliche Größe aufweisen, weil in C/C++ noch ein extra Byte für die abschließende Null benötigt wird.
- ABAP interpretiert das Stringende-Kennzeichen Null als #. C/C++ hingegen sieht das Zeichen ' ' nicht als Stringende-Kennung an.

Dies hat gravierende Folgen, wenn Strings zwischen einem SAP-System und externen C/C++-Programmen zu übertragen sind:

- Wenn ein mit einer Null initialisierter String von einem C/C++-Client an ein SAP-System gesendet wird, werden die Nullen als # interpretiert, falsch in den ABAP-String übernommen und später falsch in die Datenbanktabelle geschrieben.
- Wird umgekehrt ein String von einem SAP-System an einen externen C/C++-Server gesendet, erkennt dieser das Ende der Zeichenfolge nicht, weil das ASCII-Zeichen ' ' (Space) nicht der Stringende-Kennung in C/C++ entspricht.
- Das gravierendste Problem ist jedoch die Byteverschiebung nach rechts (String wird von einem C/C++-Programm zum SAP-System gesendet) bzw. links (String wird vom SAP-System an ein externes C/C++-Programm gesendet), die eine falsche Zuordnung der Daten bewirkt.

Um den letzten Punkt zu verdeutlichen, nehmen wir an, dass von einem externen C/C++-Client der Vor- und Nachname einer Person an ein R/3-System gesendet werden soll. Der Vor- und Nachname soll jeweils fünf Zeichen umfassen. Der C/C++-Entwickler wird in seinem Programm eine Struktur anlegen, die wie folgt definiert ist:

```
struct _PERSONENDATEN{
   char Vorname[6],
        Nachname[6];
}
```

Die Größe der Struktur ist zwölf Byte. Die entsprechende ABAP-Definition sieht jedoch so aus:

```
TYPES: BEGIN OF st_Personendaten
             , Vorname(5) TYPE c
             , Nachname(5) TYPE c
       , END OF st_Personendaten.
```

Die ABAP-Struktur ist somit nur zehn Byte groß. Sendet ein externer Client Daten vom Typ `Personendaten` an das SAP-System, geschieht das, was in Abbildung 4.1 dargestellt ist.

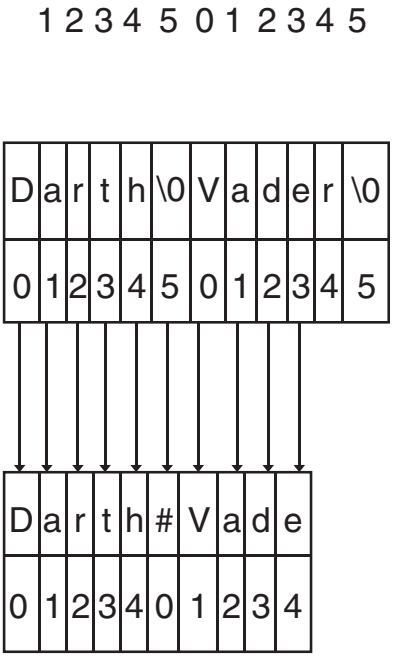


**Abbildung 4.1** Byteverschiebung nach rechts

Die Rechtsverschiebung ist deutlich zu erkennen. Das sechste Byte des Vornamens mit der Stringende-Kennung `\0` im C/C++-Client wird auf das erste Byte des Nachnamens im ABAP-Server verschoben und dort als # interpretiert, während das fünfte Byte des Nachnamens mit dem Wert `r` gar nicht mehr im ABAP-Server auftaucht. Es wird einfach abgeschnitten.

Das Problem der Byteverschiebung ist nicht nur auf Strukturen mit Zeichenketten beschränkt, sondern tritt immer dann auf, wenn in Client und Server die Daten unterschiedliche Größen besitzen.

Wie können nun die Probleme bei der Übertragung von Strings und Strukturen, die Strings enthalten, gelöst werden? Am sinnvollsten ist es, wenn Sie im C/C++-Programm mit zwei Stringvariablen oder Strukturen arbeiten. Die erste Stringvariable genügt den Anforderungen an einen String in einem C/C++-Programm, während die zweite Stringvariable den Anforderungen an einen String in einem ABAP-Programm entspricht. Dies bedeutet:

- Die Stringvariable im C/C++-Format besitzt immer eine Feldgröße, die die Summe aus der maximalen Länge der aufzunehmenden Zeichenkette plus eins für die Stringende-Kennung ist.
- Die Stringvariable im C/C++-Format wird immer mit einer Null initialisiert.
- Die Stringvariable im ABAP-Format besitzt immer eine Feldgröße, die gleich der maximalen Länge der aufzunehmenden Zeichenkette ist.
- Die Stringvariable im ABAP-Format wird immer mit ' ' (Space) initialisiert.

In einem externen Client nimmt die Stringvariable im C/C++-Format alle zu übertragenden Werte vom Anwender entgegen. Vor der Übertragung der Daten werden die Werte auf den String im ABAP-Format kopiert. Umgekehrt gilt für einen C/C++-Server, dass die Stringvariable, die den Anforderungen an das ABAP-Format genügt, die Zeichenkette als Erstes empfängt und im zweiten Schritt die Daten auf den String im C/C++-Format überträgt.

Durch dieses Vorgehen wird Folgendes sichergestellt:

- Es kommt zu keiner Byteverschiebung, weil der an das SAP-System zu sendende String im C/C++-Programm die gleiche Größe besitzt wie im SAP-Funktionsbaustein.
- Es werden keine fehlerhaften Werte für die Stringende-Kennung Null übertragen. Der String, der an das SAP-System gesendet wird, ist ja mit Space initialisiert.
- Der externe Client oder Server erkennt das Ende einer Zeichenkette immer.

### 4.1.3 Numerische Strings

Numerische Strings sind Zeichenketten, die nur aus Ziffern bestehen dürfen. In ABAP gibt es dafür den Datentyp `n`. In einem ABAP-Programm wird überprüft, ob einer Variable vom Typ `n` ein numerischer Wert zugewiesen wurde. Ist dies nicht der Fall, entfernt das Programm den nicht numerischen Wert und füllt die Variable mit führenden Nullen auf. Das nachfolgende Beispiel verdeutlicht dies:

```
REPORT ztestnumerisch.
DATA: numerisch(5) TYPE n.
START-OF-SELECTION.
numerisch = '012A4'.
WRITE: / numerisch.
```

Das Programm erzeugt die Ausgabe `00124`. Das `A` wurde entfernt und der String mit führenden Nullen aufgefüllt, bis die definierte Länge des numerischen Strings erreicht wurde.

Leider existiert in C/C++ kein numerischer Datentyp. Selbst der Datentyp `RFC_NUM` der RFC-API ist nichts anderes als ein `unsigned char`, wie aus dem folgenden Auszug der Header-Datei deutlich wird:

```
typedef unsigned char  RFC_CHAR.
typedef RFC_CHAR       RFC_NUM
```

Daher können einer Variable des Typs `RFC_NUM` alle ASCII-Zeichen zugewiesen werden. Dies umfasst auch die nicht numerischen Zeichen. Damit ergeben sich für die Übertragung von numerischen Strings zwei Fragen:

- Was geschieht, wenn einer Variablen mit Datentyp `n` in einem Funktionsbaustein ein Wert zugewiesen wird, der nicht numerisch ist?
- Wie kann sichergestellt werden, dass ein externer Client in einem numerischen Feld auch nur numerische Werte speichert?

Zur Beantwortung der ersten Frage ergänzen wir die Schnittstelle unseres Funktionsbausteins `Z_RFC_ADD` um den Importing-Parameter `IP_NUMC`, der eine Länge von fünf Zeichen besitzt. In unserem externen Client fügen wir der Schnittstellenbeschreibung ebenfalls den Parameter hinzu. Zusätzlich weisen wir im Clientprogramm dem Parameter `IP_NUMC` den Wert `012A4` zu. Abschließend aktivieren wir den ABAP Debugger, indem im Feld `TRACE` der Wert `D` eingegeben wird.

Starten wir jetzt unser C-Programm, können wir im ABAP Debugger beobachten, dass der Variablen `IP_NUMC` im Funktionsbaustein der Wert `012A4` zugewiesen wird. Das `A` wird nicht gelöscht. Statt dessen wird im SAP-Funktionsbaustein der nicht numerische Wert in einer Variablen des Datentyps `n` akzeptiert. Dieser Wert würde auch in einer Datenbanktabelle gespeichert werden.

Dieses Ergebnis ist nicht befriedigend. Die Problematik wird durch folgende Überlegung noch deutlicher. SAP pflegt in seinen Datenbanktabellen sehr häufig einen Zeitstempel, d.h., das SAP-System speichert Datum und Uhrzeit der letzten Datensatzänderung. Das Datumsfeld ist vom Typ `DATS`. Beim Datentyp `DATS` handelt es sich um ein acht Byte großes numerisches Feld. Ähnlich ist es mit der Zeit.

Hierbei handelt es sich um eine Variable des Typs `TIME`. Der Typ `TIME` ist vom Typ `n` mit der Größe sechs Byte.

In der RFC-API existieren für das Übertragen von Datums- und Zeitwerten die Typen `RFC_DATE` und `RFC_TIME`. Ihre Definition lautet:

```
typedef RFC_CHAR        RFC_DATE[8]
typedef RFC_CHAR        RFC_TIME[6].
```

Der Type `RFC_CHAR` ist, wie oben bereits erwähnt, definiert als:

```
typedef unsigned char RFC_CHAR .
```

Damit können auch in Variablen der Typen `RFC_DATE` und `RFC_TIME` alle ASCII-Zeichen gespeichert werden. Somit ist es möglich, Zeit- und Datumsvariablen in der Schnittstelle eines Funktionsbausteins nicht numerische Werte über einen externen Client zuzuweisen. Diese Werte würden auch falsch in einer Datenbanktabelle gespeichert. Dies hat weit reichende Konsequenzen. Es kann zum Beispiel nicht mehr festgestellt werden, wann welcher Datensatz geändert wurde. Daher muss in einem externen Client sichergestellt werden, dass in einer Variable numerischen Typs auch nur Ziffern gespeichert werden.

Leider verfügt die RFC-API über keine entsprechende Prüffunktion. Sie sind gezwungen, hier selbst eine zu entwickeln. Dies ist aber sehr schnell getan, wie das nachfolgende Beispiel zeigt.

```
int IsStringNumeric(char *Numc, char **pos)
{
   if(!Numc || !pos)
      return 1;
   *pos = Numc;
   while(*pos && **pos != '\0'){
      if(!isdigit(**pos))
         return 1;
      (*pos)++;
   }
   return 0;
}
```

**Listing 4.1** Beispiel für die Prüfung auf nicht-numerische Zeichen

Die Funktion `IsStringNumeric` erhält als Übergabewerte die Referenz auf den zu überprüfenden String und eine Referenz auf eine Zeigervariable vom Typ `Char`. In der letztgenannten Variable wird die Adresse des nicht numerischen Zeichens zurückgegeben, falls der String ein solches Zeichen enthält, ansonsten ist der Wert null. Die Startadresse des zu überprüfenden Strings wird der Variablen `pos`

zugewiesen. Im Anschluss wird innerhalb einer Schleife für jedes Zeichen des Strings geprüft, ob es sich um ein numerisches Zeichen handelt. Für diese Prüfung wird die Bibliotheksfunktion `isdigit` verwendet.

### 4.1.4 Die Sache mit den gepackten Zahlen

Jeder ABAP-Programmierer hat schon einmal mit gepackten Zahlen in Form von Variablen des Datentyps `Quantity`, `Currency` oder selbst definierten Variablen des Datentyps `p` zu tun gehabt. Dies sind in ABAP sehr beliebte Datentypen. Der Wert einer Variablen vom `packed`-Datentyp kann aufbereitet dargestellt werden, d.h., es werden Trennzeichen für Dezimalstellen eingefügt. Dies ist bei Variablen mit dem Datentyp `float` nicht ohne weiteres möglich.

So wird der Wert 1000,25 von einer Variablen des Typs `float` ausgegeben als `1,0002500000000000E+03`, und von einer Variablen des Typs `p decimals 2` als `1.000,25`.

Wenn aber eine Variable des Typs `p decimals` übertragen werden soll, tritt ein Problem auf: Wie lautet das entsprechende Pendant zum `packed`-Typ auf der Seite des externen Programms? Sehen Sie sich die Typen von C/C++ an, finden Sie dort keinen entsprechenden Datentyp. Daher wollen wir jetzt klären, was gepackte Zahlen sind und wie diese zwischen dem SAP-System und einem externen Programm ausgetauscht werden.

Gepackte Zahlen sind zunächst nicht anderes als *Binary Coded Decimals* (BCD). Eine gepackte Zahl wird somit in einem Feld des Datentyps `unsigned byte` dargestellt, wobei jedes Byte genau *zwei* Dezimalziffern aufnimmt. Diese Besonderheit wird dadurch ermöglicht, dass bei gepackten Zahlen jede Dezimalziffer für sich betrachtet und in eine binäre Darstellung übertragen wird.

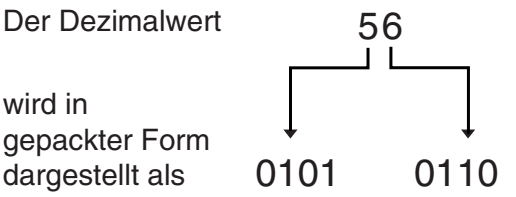


**Abbildung 4.2** Umwandlung von Dezimal- in BCD-Darstellung

Die Vierergruppe von Binärzahlen bezeichnet man auch als *Tetrade*. Da jede Tetrade vier Bit und der Datentyp `Byte` acht Bit umfasst, ist es möglich, in einem Byte zwei Tetraden (2 * 4 Bits) und somit zwei Dezimalziffern zu verwalten. Die Umwandlung einer binär kodierten Zahl in eine Dezimalziffer ist dann ganz einfach (siehe Abbildung 4.3).

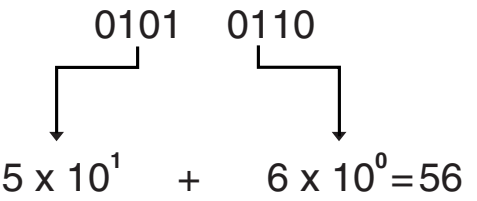


**Abbildung 4.3** Umwandlung von BCD- in Dezimaldarstellung

Welche umgewandelte Tetrade mit welcher Zehnerpotenz zu multiplizieren ist, hängt offensichtlich von der Position der Tetrade in dem Byte-Array ab.

Bleibt noch ein kleines Problem. Der Datentyp `unsigned byte` ist in der Datei *saprfc.h* der RFC-API definiert als:

```
typedef unsigned char rfc_byte_t.
```

Daher verwaltet er leider keine binären Zahlen, sondern nur Dezimalzahlen im Bereich von 0 bis 255 oder Character-Zeichen gemäß dem ASCII-Zeichensatz. Daher muss die binäre Darstellung der gepackten Zahl noch – als Ganzes – in eine Dezimalzahl umgewandelt werden. Der Vorgang ist in Abbildung 4.4 dargestellt.

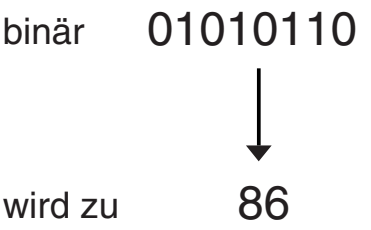


**Abbildung 4.4** Umwandlung von Dezimalwert in eine BCD-Zahl

Es sei nochmals betont, dass der Wert 86 zwei binär kodierte Ziffern, nämlich 5 und 6 repräsentiert.

Natürlich muss man als Entwickler nicht mehr selbst Routinen für die Konvertierung von gepackten Zahlen entwickeln, sondern kann hier auf folgende Funktionen der RFC-API zurückgreifen:

- `RfcConvertCharToBcd`
- `RfcConvertBcdToChar`

Die Funktion `RfcConvertCharToBcd` wandelt eine Zahl, die ihr in Form eines Strings übergeben wird, in ein entsprechendes Bytefeld um. Die Schnittstelle der Funktion ist:

| Variable | Bedeutung |
|---|---|
| `*in` | Zeiger auf einen String, der die umzuwandelnde Zahl enthält. Eine Dezimalzahl sollte der internen Darstellung einer Dezimalzahl entsprechen: Vorzeichen an erster Stelle, Dezimalstellen durch einen Punkt getrennt. |
| `in_len` | Länge des Strings |
| `*pdecs` | Referenz auf die Variable, in der die Anzahl der Dezimalstellen hinterlegt ist |
| `*out` | Referenz auf ein Feld, in dem die BCD-Darstellung der Zahl gespeichert wird |
| `out_len` | Bytegröße des Felds |
| `int` | Rückgabewert der Funktion |

**Tabelle 4.1** Schnittstelle der Funktion RfcConvertCharToBcd

Die Funktion gibt eine Variable des Datentyps `int` zurück. Über den Wert der Variablen teilt die Funktion dem rufenden Programm den Status der Ausführung mit. Es werden folgende Werte zurückgegeben:

| Rückgabewert | Bedeutung |
|---|---|
| 0 | Funktion wurde erfolgreich ausgeführt. |
| 2 | Die Größe des Felds zur Speicherung der BCD Darstellung der Zahl ist zu klein. |
| 3 | Der umzuwandelnde String enthält keine gültige Zahl. |
| 4 | Die Länge des Strings ist falsch. |

**Tabelle 4.2** Rückgabewerte der Funktion RfcConvertCharToBcd

Der Aufruf der Funktion könnte so aussehen:

```
int main(int argc, char** argv)
{
   RFC_RC rc = RFC_OK;
   rfc_byte_t Zahl[10];
   int iRC = 0,
       iDecimals = 2;
   iRC = RfcConvertCharToBcd(
                "1000.25", strlen("1000.25"),
                &iDecimals, Zahl, sizeof(Zahl));
   return (int)rc;
}
```

**Listing 4.2** Aufruf der Funktion RfcConvertCharToBcd

Das Gegenstück zur Funktion `RfcConvertCharToBcd` ist `RfcConvertBcdToChar`. Die Funktion `RfcConvertBcdToChar` wandelt ein Bytefeld in die entsprechende Dezimaldarstellung der Zahl um, wobei die Dezimaldarstellung in einem String zurückgeben wird. Der Wert in dem String kann im Anschluss unter Anwendung der Funktion `atof` einer Variablen des Typs `float` zugewiesen werden. Die Funktion `RfcConvertBcdToChar` besitzt die folgende Schnittstelle:

| Variable | Bedeutung |
|---|---|
| `*in` | Feld des Typs `RFC_BCD` mit der BCD-Darstellung der gepackten Zahl |
| `in_len` | Bytegröße des Felds |
| `decs` | Anzahl der Dezimalstellen der gepackten Zahl |
| `*out` | Referenz auf einen String, in dem die Dezimaldarstellung der gepackten Zahl zurückgegeben wird |
| `out_len` | Bytegröße des Strings |
| `int` | Rückgabewert der Funktion |

**Tabelle 4.3** Schnittstelle der Funktion RfcConvertBcdToChar

Auch bei der Funktion `RfcConvertBcdToChar` enthält der Rückgabewert Informationen über den Status der Ausführung der Funktion. Die Funktion kann folgende Werte zurückgeben:

| Rückgabewert | Bedeutung |
|---|---|
| 0 | Funktion wurde erfolgreich ausgeführt. |
| 2 | Die Länge des Strings reicht nicht aus, um in ihm die Dezimaldarstellung der gepackten Zahl zu speichern. |
| 3 | Der Wert im BCD-Feld stellt keine Zahl dar. |
| 4 | Die Größe des Bytefelds ist falsch. |

**Tabelle 4.4** Rückgabewerte der Funktion RfcConvertBcdToChar

Die Funktionen `RfcConvertCharToBcd` und `RfcConvertBcdToChar` basieren auf den Funktionen `str2nbcd` und `nbcd2str`, die vom RFC-Code angelegt werden.

Schaut man sich die Funktionen `RfcConvertCharToBcd` und `RfcConvertBcdToChar` an, ist festzustellen, dass sie einen String in eine gepackte Zahl umwandeln bzw. die Dezimalform einer gepackten Zahl in einen String zurückgeben. Daher stellt sich bei Eigenentwicklungen die Frage, ob es nicht sinnvoller ist, dass die beteiligte ABAP-Softwarekomponente die Konvertierung einer gepackten Zahl in einen String vornimmt und der externen Softwarekomponente den String übergibt.

Dies würde aber dem Gedanken, dass die an einer Client-Server-Beziehung beteiligten Softwarekomponenten weitgehend autonom sein sollen, widersprechen. In Sinne der Autonomie ist es besser, wenn die Softwarekomponenten die Daten mit ihrem originären Datentyp – sprich dem Datenbankdatentyp – austauschen. Sollte eine der beteiligten Softwarekomponenten den Datentyp nicht verarbeiten können, muss sie selbst für die Konvertierung in einen Datentyp sorgen, den sie verarbeiten kann. Sie sollte sich nicht darauf verlassen, dass ihr die Aufgabe durch die Drittsoftwarekomponente abgenommen wird.

### 4.1.5 Abschließende Übersicht über das Type-Mapping

Mit einer Übersicht über das Mapping von RFC-Datentypen auf ABAP-Datentypen und über dasjenige von RFC-Datentypen auf häufig verwendete Datenbankdatentypen werden die Betrachtungen über das Type-Mapping abgeschlossen.

| ABAP-Datentyp | Bedeutung | RFC-Datentyp |
|---|---|---|
| `c` | Zeichen (Character) | `RFC_CHAR` |
| `d` | Datum (Date, Darstellung im Format YYYYMMDD) | `RFC_DATE` |
| `f` | Gleitpunktzahl (Float) | `RFC_FLOAT` |
| `i` | ganze Zahl (Integer) | `RFC_INT` |
| `n` | numerischer String | `RFC_NUM` |
| `p` | BCD-Zahl | `RFC_BCD` |
| `string` | Zeichenfolge (String) | in Schnittstellen von Funktionsbausteinen nicht zulässig |
| `t` | Zeitpunkt (Time, Darstellung im Format HHMMSS) | `RFC_TIME` |
| `x` | Byte (Hexadecimal) | `RFC_BYTE` |
| `xstring` | Bytefolge (X-String) | in Schnittstellen von Funktionsbausteinen nicht zulässig |

**Tabelle 4.5** Mapping zwischen ABAP- und RFC-Datentypen

| Datenbank-datentyp | Bedeutung | RFC-Datentyp |
|---|---|---|
| `ACCP` | Buchungsperiode JJJJMM | `RFC_CHAR` |
| `CHAR` | Zeichenfolge | `RFC_CHAR` |

**Tabelle 4.6** Mapping zwischen Datenbank- und RFC-Datentypen

| Datenbank-datentyp | Bedeutung | RFC-Datentyp |
|---|---|---|
| CLNT | Mandant | RFC_CHAR |
| CUKY | Währungsschlüssel, wird von CURR-Feldern referiert | RFC_CHAR |
| CURR | Währungsfeld, abgelegt als DEC | RFC_BCD |
| DATS | Datumsfeld (JJJJMMDD), abgelegt als CHAR(8) | RFC_DATE |
| DEC | Rechen- oder Betragsfeld mit Komma und Vorzeichen | RFC_BCD |
| FLTP | Gleitpunktzahl mit acht Byte Genauigkeit | RFC_FLOAT |
| INT1 | 1-Byte-Integer | RFC_INT1 |
| INT2 | 2-Byte-Integer, nur für Längenfeld vor LCHR oder LRAW | RFC_INT2 |
| INT4 | 4-Byte-Integer | RFC_INT |
| LANG | Sprachkennzeichen | RFC_CHAR |
| NUMC | numerische Zeichenfolge | RFC_NUM |
| QUAN | Mengenfeld, abgelegt als DEC | RFC_BCD |
| RAW | uninterpretierte Folge von Bytes | RFC_BYTE |
| TIMS | Zeitfeld (HHMMSS), abgelegt als CHAR(6) | RFC_TIME |
| UNIT | Einheitenschlüssel für QUAN-Felder | RFC_CHAR |

**Tabelle 4.6** Mapping zwischen Datenbank- und RFC-Datentypen (Forts.)

## 4.2 Der Umgang mit Strukturen

Unsere bisherigen externen Client- und Serverprogramme waren recht einfach gestrickt, da sie auch nur einfache Datentypen mit dem R/3-System austauschen. Leider werden in der Realität nur sehr selten einzelne Variablen ausgetauscht. Statt dessen werden meistens ein oder mehrere Datensätze (Strukturen) gesendet. Wie Sie Datensätze austauschen können, wird im Folgenden dargelegt.

Das Übertragen einer Struktur wird anhand eines Beispielprogramms vorgestellt. Das Ziel ist die Übertragung eines Kundenstammdatensatzes von einem R/3-System zu einem externen Programm. Im Clientprogramm wird die Kundennummer des Kunden erfasst, für den die Stammdaten aus dem R/3-System zu lesen und an den Client zu übertragen sind.

Für den SAP-Teil des Beispiels verwenden wir den Funktionsbaustein Z_RFC_GET_SINGLE_CUSTOMER, der bereits vorgestellt wurde. Wir müssen somit nur

noch das externe Programm entwickeln. Damit ein externes Programm mit einem Funktionsbaustein eine Strukturvariable austauschen kann, müssen in diesem Programm folgende zusätzliche Schritte ausgeführt werden:

- Im Programm muss eine Strukturdefinition angelegt werden. Die Definition der Struktur muss mit der Strukturdefinition im R/3-System übereinstimmen.
- Die Beschreibung des Strukturaufbaus muss bei der RFC-API registriert werden.
- Die Strukturvariable muss als Parameter in der Beschreibung der Schnittstelle für den Datenaustausch bekannt gegeben werden.

Auf unser Beispiel bezogen bedeutet dies, dass wir die Struktur `_RSCUSTOMERSAP` anlegen. Sie besitzt den gleichen Aufbau wie die Datenbanktabelle ZKNA1JMS im Data Dictionary. Die Struktur wird benutzt, um die Kundenstammdaten mit dem R/3-System auszutauschen. Aufgrund der geschilderten Probleme mit Strings verwenden wir aber noch eine zweite Strukturdefinition. Die Struktur `_RSCUSTOMEREXT` besitzt die gleichen Felder wie die Struktur `_RSCUSTOMERSAP`, jedoch sind die `CHAR`-Felder alle um ein Byte größer, sodass auch das Stringende-Kennzeichen `\0` gespeichert werden kann. Die Struktur `_RSCUSTOMEREXT` wird verwendet, um die empfangenen Daten weiterzuverarbeiten. Die Struktur `_RSCUSTOMERSAP` ist definiert als:

```
typedef struct _RSCUSTOMERSAP
{
   RRC_CHAR MANDT[3];
   RFC_CHAR KUNNR[10];
   RFC_CHAR NAME1[35];
   RFC_CHAR LAND1[3];
   RFC_CHAR ORT01[35];
   RFC_CHAR PSTLZ[10];
   RFC_CHAR STRAS[35];
   RFC_INT  UMSEXP;
   RFC_CHAR WAERS[5];
   RFC_CHAR TELF1[16];
   RFC_CHAR TELFX[31];
} RSCUSTOMERSAP;
```

**Listing 4.3** Definition der Struktur _RSCUSTOMERSAP

Als Nächstes müssen wir eine Beschreibung des Strukturaufbaus anlegen und bei der RFC-API registrieren. Das Anlegen einer Strukturbeschreibung beinhaltet, dass für jedes Feld der Struktur eine Beschreibung der technischen Eigenschaften erfasst wird. Die technischen Eigenschaften werden dabei der Struktur im Data

Dictionary entnommen. Die Beschreibung wird in einem Array des Typs RFC_TYPE_ELEMENT2 hinterlegt. Dieser Typ ist in der RFC-API definiert als:

| Feld | Bedeutung |
|---|---|
| name | Name des Felds in der Data-Dictionary-Struktur des R/3-Systems |
| type | Typ des Felds. Hier ist ein Wert des Aufzähltyps RFC_TYPE zu hinterlegen. |
| length | Bytegröße des Felds |
| decimals | Anzahl der Dezimalstellen |
| offset | Distanz des Felds vom Beginn der Struktur in Byte |

**Tabelle 4.7** Aufbau der Struktur RFC_TYPE_ELEMENT2

Der Offset für ein Feld kann so berechnet werden:

$$\text{Offset}_{\text{akt Feld}} = \text{Offset}_{\text{Vorgänger}} + \text{Bytegröße des Datentyps vom Vorgängerfeld}$$

Sofern sich die Struktur in dem externen Programm auf eine Struktur im Data Dictionary bezieht, existieren folgende Optionen für die Ermittlung der Bytegröße des Datentyps:

- In der Datenbanktabelle DD03L steht in dem Feld INTLEN für jedes Feld einer Struktur oder Tabelle die Bytegröße des zugrunde liegenden Datentyps.
- Im Data Dictionary können Sie sich über das Menü **Hilfsmittel · Laufzeitobjekt anzeigen** u.a. eine Übersicht über die Eigenschaften der Felder einer Struktur oder Tabelle ansehen. In der Spalte DDLn ist die Bytegröße des Datentyps zu sehen (siehe Abbildung 4.5).

Ferner kann dem Laufzeitobjekt für ein Data-Dictionary-Element auch direkt der Offset für ein Feld entnommen werden. Er steht in der Spalte Offs.

Für das Feld NAME1 der Struktur _RSCUSTOMERSAP sieht der Eintrag so aus:

```
RFC_TYPE_ELEMENT2 DESC_RS_CUSTOMER_SAP[11]
DESC_RS_CUSTOMER_SAP[2].name =     "NAME1"
DESC_RS_CUSTOMER_SAP[2].type =     TYPC
DESC_RS_CUSTOMER_SAP[2].length =   35;
DESC_RS_CUSTOMER_SAP[2].decimals = 0;
DESC_RS_CUSTOMER_SAP[2].offset   =
                       DESC_RS_CUSTOMER_SAP[1].offset
                     + DESC_RS_CUSTOMER_SAP[1].length;
```

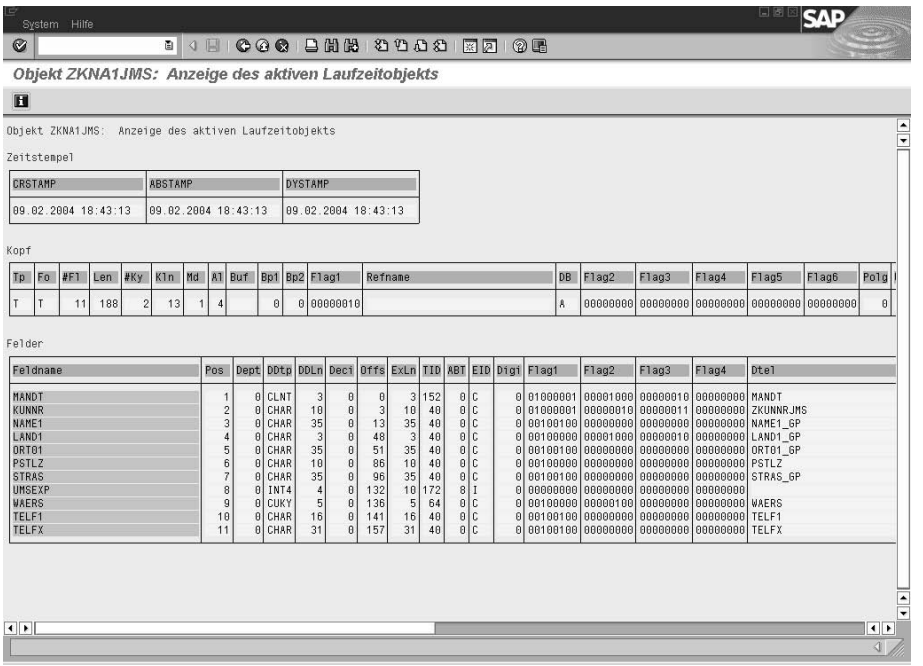


**Abbildung 4.5** Laufzeitobjekt der Tabelle ZKNA1JMS

Die Größe des Arrays entspricht der Anzahl der Felder in der Struktur. Unser Array muss elf Beschreibungen speichern, da die Tabelle ZKNA1JMS elf Felder umfasst. Das Anlegen der Strukturbeschreibung kann durch folgende Hilfsfunktion vereinfacht werden:

```
RFC_TYPE_ELEMENT2 BuildTableElement(char* pName,
                                    int   iType,
                                    int   iLength,
                                    int   iDecimals
                                    int   iOffset)
{
   RFC_TYPE_ELEMENT2 Element;
   Element.name     = pName;
   Element.type     = iType;
   Element.length   = iLength;
   Element.decimals = iDecimals;
   Element.offset   = iOffset;
   return Element;
};
```

**Listing 4.4** Hilfsfunktion, um die Struktur RFC_TYPE_ELEMENT2 zu füllen

Der Aufruf der Funktion sieht für das Feld NAME1 der Tabelle ZKNA1JMS so aus:

```
DESC_RS_CUSTOMER_SAP[2] = BuildTableElement("NAME1",
                          TYPC, 35, 0,
                        ( DESC_RS_CUSTOMER_SAP[1].offset
                       + DESC_RS_CUSTOMER_SAP[1].length) );
```

Im Anschluss an den Aufbau der Feldbeschreibung muss dieser gegenüber der RFC-API registriert werden. Dies geschieht durch die Funktion RfcInstallStructure2. Der Funktion müssen Werte für nachfolgende Variablen übergeben werden:

| Variable | Bedeutung |
|---|---|
| name | Name der Struktur, mit der die Beschreibung verknüpft wird |
| elements | Adresse des Arrays mit der Beschreibung des Feldaufbaus |
| entries | Anzahl der Elemente in dem Feld |
| pTypeHandle | Referenz auf eine Variable des Typs RFC_TYPEHANDLE. Die Variable nimmt das Type-Handle auf, mit dem die RFC-API die Feldbeschreibung verknüpft hat. |
| RFC_RC | Rückgabewert der Funkion, der den Status der Ausführung signalisiert |

**Tabelle 4.8** Schnittstelle der Funktion RfcInstallStructure2

Wenn die Funktion RfcInstallStructure2 erfolgreich ausgeführt wurde, steht in der Variablen pTypeHandle dasjenige Handle, mit dem die RFC-API die Strukturbeschreibung verknüpft hat.

Zum Schluss muss die Strukturvariable RSCustomerSAP vom Typ RSCUSTOMERSAP in der Schnittstellenbeschreibung bekannt gegeben werden. Dies geschieht durch das Eintragen der Strukturvariable in das Array ExpParam. Das Array nimmt alle Variablen auf, für die unser Programm Daten vom R/3-System empfangen kann. Wichtig ist, dass wir jetzt das durch die Funktion RfcInstallStructure2 ermittelte Type-Handle im Feld type der Struktur RFC_PARAMETER übergeben.

Ruft das Clientprogramm jetzt den Funktionsbaustein Z_RFC_GET_SINGLE_CUSTOMER im R/3-System auf, steht in der Strukturvariable RSCustomerSAP der Kundenstammdatensatz.

Das unten stehende Programmfragment ist ein Ausschnitt aus dem Programm GetCustomerSAP. Es zeigt nochmals die genannten Schritte, um eine Strukturvariable als Austauschparameter anzulegen.

```
RFC_RC GET_CUSTOMER_FROM_SAP(RFC_HANDLE hConnection,
                         RFC_ERROR_INFO_EX *pErrorInfo)
{
   / *Deklaration der Variablen zum Aufnehmen der
     Daten von bzw. zum Senden an SAP */
   :
   RSCUSTOMERSAP RSCustomerSAP;
   /* Deklaration der Arrays für die Importing- und
     Exporting-Parameter */
   RFC_PARAMETER ImpParam[2]
               , ExpParam[2]
               ;
   /* zusaetzliche Deklarationen fuer Struktur */
   RFC_TYPEHANDLE   Tabletype;
   RFC_TYPE_ELEMENT2 DESC_RS_CUSTOMER_SAP[11];
   :
   /* Aufbau der Feldbeschreibung für die Struktur
     _RSCUSTOMERSAP */
   DESC_RS_CUSTOMER_SAP[0] = BuildTableElement
                                  ("MANDT", TYPC, 3, 0, 0);
   :
   /* Struktur bei der RFC-API registrieren */
   if(( rc = RfcInstallStructure2
           ("IS_CUSTOMER_DATA", DESC_RS_CUSTOMER_SAP,
            11, &Tabletype)) != RFC_OK){
        RfcLastErrorEx(pErrorInfo);
        return RFC_FAILURE;
   };
   /* Strukturvariable als Exporting-Parameter bekannt-
    geben */
   ImpParam[0] = BuildSimpleParam("IS_CUSTOMER_DATA",
   /* Tabletype der Funktion RfcInstallStructure2 ueber-
   geben */
                                    Tabletype,
                                    sizeof(RSCUSTOMERSAP),
                                    &RSCustomerSAP);
```

**Listing 4.5** Anlegen einer Struktur als Importing-Parameter

Sie werden zurecht kritisch anmerken, dass das manuelle Anlegen der Beschreibung des Feldaufbaus für umfangreiche Tabellen und Strukturen sehr zeitintensiv ist. Daher wird in Kapitel 6 eine Lösung zur Automatisierung vorgestellt. An dieser

Stelle wollen wir jedoch noch veranschaulichen, wieso diese Arbeit vor allem in komplexeren Systemumgebungen unumgänglich ist. Dazu müssen wir noch einmal auf die oben beschriebenen Abbildungen des ABAP-Typsystems auf das RFC-System eingehen:

- Das ABAP-Typsystem wird vom SAP-Applikationsserver bereitgestellt und ist damit auf einer abstrakten Ebene plattformunabhängig.
- Die Plattformen, auf denen z. B. die Clients ausgeführt werden, haben jedoch z.T. unterschiedliche Formen in der internen Darstellung von Daten. Hierzu gehören natürlich die Größe der einzelnen Datentypen (z. B. haben 32-Bit-Plattformen einen vier Byte großen `Long`-Typ) und die Anforderungen an die Ausrichtung (Alignment) von Daten. Es kann z. B. sein, dass eine Plattform fordert, dass Integer-Werte immer auf 4- oder gar auf 8-Byte-Grenzen ausgerichtet sind.

Für den Bereich der Netzwerk-Programmierung, die die RFC-Entwicklung ja letztlich ist, kommt zudem noch die so genannte *Endian*-Problematik hinzu. Die Grundentscheidung, die hier eine Rolle spielt und auf der Hardware-Ebene einer Plattform getroffen wird, betrifft die Anordnung der einzelnen Bytes innerhalb einer Mehr-Byte-Variablen, z. B. einer 4-Byte-Integer. Auf vielen Plattformen wird die auf den ersten Blick seltsam anmutende Form gewählt, dass das »unwichtigste« Byte an vorderer Stelle im Speicher auftaucht (»Little Endian«). Dies ist z. B. bei den Intel-Prozessoren der x86-Reihe so. Andere CPUs hingegen, wie z. B. die SUN-Sparc-Serie, speichern das wichtigste, d. h. höchstwertige Byte auch an vorderer Stelle im Speicher ab. Diese zunächst unscheinbare Tatsache kann allerdings z. B. bei der Übertragung gemischter Datentypen – wie eben Strukturen – erhebliche Konsequenzen haben. Das folgende Beispiel zeigt zunächst die Big-Endian-Darstellung einer Struktur aus einer Integer-Variablen (hier mit Wert 4) und einer Zeichenvariablen (Wert 'a'):

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 4 | 'a' |

Jedes Feld entspricht dabei einem Byte. In einer Little-Endian-Umgebung wäre das entsprechende Layout im Speicher:

| | | | | |
|---|---|---|---|---|
| 4 | 0 | 0 | 0 | 'a' |

Die Konvertierung einer solchen einfachen Struktur von einem Little-Endian-System zu einem Big-Endian-System gelingt damit offensichtlich nur noch, wenn Informationen über das Speicher-Layout mitgeliefert werden. Gerade diese Informationen sind es, die bei der Anmeldung einer Struktur bei der RFC-API mitgegeben werden.

## 4.3 Der Umgang mit internen Tabellen

Interne Tabellen werden eingesetzt, wenn zwischen einem externen Programm und einem SAP-Funktionsbaustein mehrere Datensätze auszutauschen sind. Dies ist sehr häufig der Fall. Somit sind Kenntnisse im Umgang mit internen Tabellen auch für die RFC-Programmierung wichtig.

Um Ihnen den Umgang mit internen Tabellen zu demonstrieren, wird ein Modul entwickelt, das in der Lage ist, neue Kundenstammdaten an den Funktionsbaustein `Z_RFC_CREATE_NEW_CUSTOMER` zu schicken. Der Funktionsbaustein schreibt die Datensätze auf der Datenbanktabelle ZKNA1JMS fort.

Überlegen wir uns zunächst das Design des Moduls. Das Modul soll die Kundenstammdatensätze inklusive ihrer Übertragung an das R/3-System verwalten. Es muss daher intern den Aufbau der Schnittstelle für den Austausch der Daten mit dem SAP-Funktionsbaustein kapseln. Ferner muss das Modul Operationen zur Manipulation der Datensätze von außen bereitstellen. Diese Funktionen für die Manipulation der Datensätze sind die eigentlichen Schnittstellen dieses Moduls. Zusätzlich muss das Modul die Definition der Struktur der Datensätze verfügbar machen. Durch die Definition wissen Programmteile, die die Operationen des Moduls aufrufen, wie die zu übergebenden Datensätze aussehen müssen. Zum Abschluss wird eine Operation benötigt, die die Daten an das SAP-System überträgt.

An dieser Stelle beschränken wir uns auf die Realisierung der Funktionalität in C, um die Konsistenz zur bisherigen Darstellung zu wahren. Sicher liegt hier die Verwendung von C++ nahe. Eine entsprechende Implementierung in C++ finden Sie auf der zum Buch gehörenden Webseite auf *www.sap-press.de*.

Ein Designentwurf könnte wie folgt aussehen:

```
/* öffentliche Funktionen */
  RFC_RC Initialisation(void);
  RFC_RC AppendCustomer(RSCUSTOMEREXT  *pCustomerExt,
                        RFC_ERROR_INFO_EX *pErrorInfo);
  RFC_RC InsertCustomer(char *KUNNR,
                        RSCUSTOMEREXT *pCustomerExt,
                        RFC_ERROR_INFO_EX *pErrorInfo);
  RFC_RC UpdateCustomer(char *KUNNR,
                        RSCUSTOMEREXT *pCustomerExt,
                        RFC_ERROR_INFO_EX *pErrorInfo);
  RFC_RC DeleteCustomer(char *KUNNR,
                        RFC_ERROR_INFO_EX *pErrorInfo);
```

```
  RFC_RC GetCustomer(char *KUNNR,
                     RSCUSTOMEREXT* pCustomerExt,
                     RFC_ERROR_INFO_EX *pErrorInfo);
  RFC_RC SendDataToSAP(RFC_HANDLE *phConnection,
                       RFC_ERROR_INFO_EX *pErrorInfo);
  int GetIndexOfKunnr(char *KUNNR);
  int GetAnzRS(void)
      {return ItFill(m_TableParam[0].ithandle);};

/* modul-interne Funktionen */
  void InitialMembers(void);
  void CreateInterface(void);
  RSCUSTOMERSAP ConvertDataFromExtToSAPFormat(
                          RSCUSTOMEREXT *pCustomerExt);
  RSCUSTOMEREXT ConvertDataFromSAPToExtFormat(
                          RSCUSTOMERSAP *pCustomerSAP);
/* modul-interne Members */
/* für die Tabellen-Parameter */
  static RFC_TABLE m_TableParam[2];
/* Verwalten von Kundennummer u. Tabellenindex */
typedef struct
{
   char Kunnr[11];
   int iTableIndex;
}RSINDEXKUNNR;
/* Array zur Verwaltung der Kundennummer und ihres
   Index in der internen Tabelle */
RSINDEXKUNNR RSIndexKunnr[600];
```

**Listing 4.6** Definitionen der Komponenten des Moduls CreateCustomerSAP zum Austausch von Kundenstammdaten

Der oben vorgestellte Ansatz unterscheidet zwischen öffentlichen und modul-internen Funktionen und Variablen. Durch die Bezeichnung »modul-intern« wird zum Ausdruck gebracht, dass die Funktionen nicht durch einen Client, der das Modul verwendet, benutzt werden können. Würde der oben vorgestellte Ansatz unter Windows als DLL umgesetzt, wären modul-interne Funktionen nicht in der Definitionsdatei enthalten. Die Definitionsdatei sähe demnach folgendermaßen aus:

```
; RfcCreateCustomer.def Definitionsdatei für
; exportierte Funktionen
LIBRARY      "RfcCreateCustomer"
```

```
DESCRIPTION  "Bibliothek-Funktionen zur Kundenanlage"

EXPORTS
   Initialisation
   AppendCustomer
   InsertCustomer
   UpdateCustomer
   DeleteCustomer
   GetCustomer
   SendDataToSAP
   GetIndexOfKunnr
   GetAnzRS
```

**Listing 4.7** Definitionsdatei unter Windows für das Modul CreateCustomerSAP

Im obigen Entwurf werden die auszutauschenden Tabellen im Array `m_TableParam` gekapselt. Die Funktion `Initialisation` dient zur Initialisierung des Moduls. Sie ruft intern die Funktionen `InitialMembers` und `CreateInterface` für die Initialisierung des Moduls auf. Die Funktion `CreateInterface` baut die Beschreibung der Schnittstelle zum SAP-Funktionsbaustein auf. Für den Client ist es bequemer, über die Kundennummer auf den Kundenstammdatensatz zuzugreifen. Deshalb wurde die modul-interne Variable `RSIndexKunnr` eingeführt. Die Variable referiert auf die Struktur `RSINDEXKUNNR`. In der Struktur wird die Zuordnung von Kundennummer zum entsprechenden Tabellenindex verwaltet. Somit ist es möglich, dass den Funktionen `AppendCustomer`, `InsertCustomer`, `UpdateCustomer`, `DeleteCustomer` und `GetCustomer`, die den Zugriff auf die Datensätze realisieren, an ihrer Schnittstelle die Kundennummer übergeben wird. Die Funktion `SendDataToSAP` bietet die Möglichkeit, die Daten an das SAP-System zu senden. Zum Schluss kann durch die Funktion `GetIndexOfKunnr` der Tabellenindex zu einer Kundennummer ermittelt werden.

### 4.3.1 Anlegen einer internen Tabelle in einem externen Programm

Alle Funktionen der RFC-API für die Arbeit mit internen Tabellen sind in der Datei *sapitab.h* definiert. Daher ist diese Datei zusätzlich in ein externes Programm einzubinden, das mit internen Tabellen arbeitet.

Um eine interne Tabelle in einem externen Programm anzulegen, sind folgende Schritte erforderlich:

- Reservieren von Speicher für die interne Tabelle. In dem angeforderten Speicherbereich werden die Datensätze gespeichert.

- Anlegen und Registrieren des Strukturaufbaus der Tabelle gegenüber der RFC-API
- Einfügen des Tabellenparameters in die Schnittstellenbeschreibung für den Datenaustausch

Die Allokation des Speichers erfolgt durch die Funktion `ItCreate`. Der Funktion müssen Werte für die folgenden Variablen übergeben werden:

| Variable | Bedeutung |
|---|---|
| `name` | Name, um die interne Tabelle in Trace-Dateien zu identifizieren |
| `leng` | Bytegröße der Struktur. Sie wird mit dem `sizeof`-Operator ermittelt. |
| `occu` | Anzahl der Datensätze, für die Speicher angefordert werden soll, wenn erstmalig ein Datensatz in die interne Tabelle eingefügt wird<br>Speichergröße = Anzahl der Datensätze * Bytegröße der Struktur |
| `memo` | nur intern von SAP genutzt, muss mit 0 besetzt werden |
| `ITAB_H` | Das Handle der Tabelle ist der Rückgabewert der Funktion. |

**Tabelle 4.9** Schnittstelle der Funktion ItCreate

Wurde die Funktion `ItCreate` erfolgreich ausgeführt, gibt sie das Handle auf die interne Tabelle zurück. Das Handle bildet eine symbolische Verknüpfung zu dem Speicher, der für die Datensätze reserviert wurde. Seine Bedeutung veranschaulicht Abbildung 4.6.

Es wird deutlich, dass das externe Programm den Speicherbereich mit den Datensätzen der internen Tabelle nicht direkt verwaltet. Nur die RFC-API hat diesen Zugriff auf den Speicherbereich. Möchte das externe Programm auf Datensätze in diesem Speicherbereich zugreifen, geht das nur über die Funktionen der RFC-API. Dazu muss das externe Programm das Tabellenhandle übergeben, durch das die RFC-API den Speicherbereich identifiziert.

Das Pendant zur Funktion `ItCreate` ist die Funktion

```
int SAP_API ItDelete (ITAB_H itab)
```

Die Funktion `ItDelete` löscht alle Informationen bezüglich der durch das Tabellenhandle identifizierten internen Tabelle. Die Funktion gibt den Speicherplatz, den die Datensätze belegen, wieder frei. Danach wird der Eintrag für die Verwaltungsinformationen der internen Tabelle gelöscht. Das Tabellenhandle ist nach der Ausführung der Funktion nicht mehr zu benutzen.

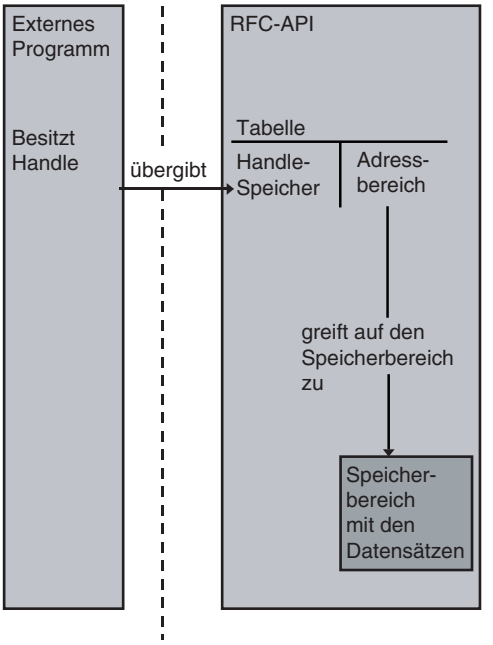


**Abbildung 4.6** Bedeutung des Tabellenhandles

Soll nur der Speicherplatz freigegeben werden, den die Datensätze in der internen Tabelle belegen, ist folgende Funktion der RFC-API anzuwenden:

```
int SAP_API ItFree (ITAB_H itab)
```

Sie gibt nur den Speicherbereich frei, den die Datensätze belegen. Aber die Verwaltungsinformationen der internen Tabelle werden nicht gelöscht. Die interne Tabelle kann somit weiter genutzt werden, um zum Beispiel neue Zeilen anzuhängen.

Nachdem die interne Tabelle angelegt wurde, müssen wir als Nächstes die Beschreibung des Strukturaufbaus bei der RFC-API registrieren. Das Anlegen der Beschreibung des Feldaufbaus für Tabellen und seine Registrierung ist identisch mit dem im vorigen Abschnitt geschilderten Verfahren für Strukturen. Es wird daher hier nicht wiederholt.

Abschließend müssen wir noch die Schnittstellenbeschreibung für den Datenaustausch um die interne Tabelle ergänzen. Alle mit dem SAP-System auszutauschenden Tabellen werden in einem Array des Typs `RFC_TABLE` verwaltet. Bei diesem Typ handelt es sich um eine Struktur der RFC-API. Sie verfügt über den folgenden Aufbau:

| Feld | Bedeutung |
|---|---|
| name | Name der Tabelle in der Schnittstelle des Funktionsbausteins |
| nlen | Länge des Namens der Tabelle |
| type | Handle auf die Beschreibung des Strukturaufbaus, mit dem die Tabelle verknüpft ist. Das Handle wird durch die Funktion RfcInstallStructure2 ermittelt. |
| leng | Bytegröße der Struktur |
| ithandle | Tabellenhandle – Rückgabewert der Funktion ItCreate |
| itmode | Art der Datenübergabe zwischen SAP-System und externem Programm |
| newitab | Flag zur Kennzeichnung, dass die Tabelle durch die RFC-API angelegt wurde |

**Tabelle 4.10** Aufbau der Struktur RFC_TABLE

Die Struktur RFC_TABLE wird für unser Beispiel in der Funktion CreateInterface mit folgenden Werten belegt:

```
m_TableParam[0].name   = "IT_CUSTOMER_DATA";
m_TableParam[0].nlen   = strlen("IT_CUSTOMER_DATA");
m_TableParam[0].type   = Tabletype;
m_TableParam[0].leng   = sizeof(_RSCUSTOMERSAP);
m_TableParam[0].itmode = RFC_ITMODE_BYREFERENCE;
m_TableParam[0].ithandle =
   ItCreate("IT_CUSTOMER_DATA", sizeof(_RSCUSTOMERSAP),
            100, 0);
```

**Listing 4.8** Aufbau der Schnittstellenbeschreibung für die Tabelle IT_CUSTOMER_DATA

Die Art der Übergabe der internen Tabelle wird durch den Wert im Feld itmode spezifiziert. Hier können folgende Werte angegeben werden:

| Wert | Bedeutung |
|---|---|
| RFC_ITMODE_BYREFERENCE | Die Übergabe erfolgt als Referenzübergabe. Änderungen am Tabelleninhalt werden dem Client übermittelt. |
| RFC_ITMODE_BYVALUE | Die Übergabe erfolgt als Wertübergabe. Änderungen am Tabelleninhalt werden dem Client nicht übermittelt. Diese Art der Übergabe wird nur SAP-intern verwendet. |
| RFC_ITMODE_KEEPALIVE | Diese Art der Übergabe wird nur SAP-intern verwendet. |

**Tabelle 4.11** Mögliche Werte des Felds itmode

### 4.3.2 Verwaltung von Datensätzen in einer internen Tabelle

Dieser Abschnitt beschreibt, wie die RFC-API die Datensätze in einer internen Tabelle verwalten könnte. Zwar muss sich der Entwickler nicht mehr um die Programmierung der Verwaltung kümmern, dennoch sind Kenntnisse über die Interna einer internen Tabelle wichtig für den richtigen Umgang mit ihr.

Eine interne Tabelle muss in der Lage sein, Datensätze mit unterschiedlichem Aufbau und unterschiedlicher Größe zu verwalten. Schließlich können nicht nur Kundenstammdaten an das SAP-System gesandt werden, sondern auch Auftragsdaten, die sicher einen anderen Aufbau als Kundenstammdaten besitzen. Die Verwaltung von Daten mit unterschiedlichem Aufbau über eine gemeinsame Programmierschnittstelle erfolgt am effektivsten in einem Feld des Datentyps `void*` – sprich: Zeiger des Typs `void`. Er besitzt die Eigenschaft, dass er die Adresse einer Variablen mit einem beliebigen Datentyp speichern kann. Er erfüllt somit genau die Bedingungen des von uns gesuchten Datentyps.

Eine Funktion zur Initialisierung eines Arrays vom Typ `void*` könnte wie unten dargestellt programmiert sein.

```
typedef void* PVOID;
PVOID *ArrayAnyType;
int iRSSize = 0;
void CreateArrayAnyType(int iAnzArgu, int iSizeOfRS)
{
   ArrayAnyType = (void**)malloc(iAnzArgu);
   iRSSize = iSizeOfRS;
   for(int iArgC = 0; iArgC < iAnzArgu; iArgC++){
      ArrayAnyType[iArgC] = NULL;
   }
}
```

**Listing 4.9** Funktion zur Initialisierung eines Arrays des Typs void*

Die Funktion `CreateArrayAnyType` erhält als Übergabevariable die Anzahl der anzulegenden Datensätze sowie die Bytegröße des Datensatzes, der in ihr gespeichert werden soll. Sie legt ein Feld von Zeigern des Typs `void*` an. Die Zeiger des Typs `void` nehmen später die Adresse des neu reservierten Speichers für die einzelnen Datensätze auf. Ferner wird die Bytegröße des Datensatzes in der globalen Variable `iRSSize` gespeichert. Zum Schluss wird in einer Schleife der Inhalt des Felds mit `NULL` initialisiert.

Wird die Variable `iAnzArgu` durch die Variable `occu` und die Variable `iSizeOfRS` durch die Variable `leng` der Funktion `ItCreate` ersetzt, wird deutlich, dass die

Funktion `CreateArrayAnyType` die Funktion `ItCreate` der RFC-API simuliert. Es fehlt lediglich die Variable `name`, da keine Einträge in eine Trace-Datei geschrieben werden. Der Aufruf der Funktion `CreateArrayAnyType` sieht unter der Annahme, dass Datensätze des Typs `RSCUSTOMERSAP` gespeichert werden sollen, wie folgt aus:

```
CreateArrayAnyType(10, sizeof(RSCUSTOMERSAP));
```

Das Speichern und Auslesen von Datensätzen in dem Array zeigen die Funktionen `InsertRS` und `GetRS`.

```
void* InsertRS(int iIndex)
{
   return ArrayAnyType[iIndex] = calloc(1, iRSSize);
};
void* GetRS(int iIndex)
{
   return ArrayAnyType[iIndex];
};
```

Die Funktion `InsertRS` erhält den Index zur Identifikation des Felds, in dem die Adresse des neu angeforderten Speichers verwaltet werden soll. Sie reserviert Speicher für den neu anzulegenden Datensatz über die Funktion `calloc`. Die Adresse der Speicherstelle wird zurückgegeben, sodass der Aufrufer der Funktion `InsertRS` den Inhalt der Speicherstelle manipulieren kann. Der Aufruf der Funktion sieht wie folgt aus:

```
RSCUSTOMERSAP *pCustomer = NULL;
pCustomer = (RSCUSTOMER*)InsertRS(2);
```

Die Funktion `InsertRS` zeigt auch Ähnlichkeit mit der Funktion `ItInsLine` der RFC-API, die weiter unten noch besprochen wird.

Die Funktion `GetRS` zeigt zum Abschluss den Zugriff auf einen Datensatz. Der Zugriff erfolgt über den Index zur Identifizierung des Felds, in dem die Adresse des Datensatzes gespeichert ist. Die Funktion `GetRS` gibt die Adresse der Speicherstelle zurück. Auch diese Funktion weist Ähnlichkeit mit der Funktion `ItGupLine` der RFC-API auf.

Fassen wir die Erkenntnisse nochmals kurz zusammen:

- Eine interne Tabelle kann mit einem Array vom Typ `void*` verglichen werden.
- Das Anlegen einer internen Tabelle kann mit dem Anfordern von Speicher für das Array des Typs `void*` verglichen werden.

- Im Array werden die Adressen der Speicherstellen verwaltet, an denen sich die Datensätze befinden.
- Den Zugriff auf die Adresse eines Datensatzes erhalten Sie über den Index.
- Ein neuer Datensatz wird in das Feld in der Art eingefügt, dass zur Laufzeit Speicher für den Datensatz reserviert wird. Die Adresse der Speicherstelle wird in der Variablen des Typs `void*` gespeichert. Diese ist ein Element des Arrays vom Typ `void*` und kann über ihren Index angesprochen werden.

### 4.3.3 Schreiben und Lesen von Datensätzen in eine interne Tabelle

Im vorangegangenen Abschnitt wurde die Verwaltung einer internen Tabelle simuliert, um ein besseres Verständnis des Begriffs »interne Tabelle« zu erlangen. Jetzt wollen wir die Funktionen der RFC-API für die Verwaltung von Datensätzen in internen Tabellen näher betrachten. Damit kehren wir wieder zu unserem Modul `CreateCustomerSAP` zurück, dessen Aufgabe die Verwaltung von Kundenstammdaten ist.

Das Modul sieht Operationen für den Zugriff auf die Kundenstammdaten vor. Dies sind die Funktionen `AppendCustomer`, `InsertCustomer`, `UpdateCustomer`, `DeleteCustomer` und `GetCustomer`. Die Funktionen `AppendCustomer` und `InsertCustomer` fügen neue Kundenstammdaten in die interne Tabelle ein. Der Unterschied zwischen den beiden Funktionen besteht darin, dass `AppendCustomer` den neuen Datensatz an das Ende der internen Tabelle anhängt, während `InsertCustomer` den Datensatz an einer bestimmten Position einfügt. Die Funktion `UpdateCustomer` aktualisiert einen existierenden Datensatz und `DeleteCustomer` löscht einen Datensatz. `GetCustomer` gibt einfach die Kundenstammdaten zurück. Diese Funktionen werden mit denen der RFC-API für das Verwalten von Datensätzen in internen Tabellen realisiert.

Um die Methoden `AppendCustomer` und `InsertCustomer` zu realisieren, werden die Funktionen `ItAppLine` und `ItInsLine` der RFC-API benötigt. Die Definition der Funktion `ItAppLine` lautet

```
void* SAP_API ItAppLine(ITAB_H itab);
```

Die Funktion `ItAppLine` hängt einen Datensatz an das Ende einer internen Tabelle an. Sie benötigt als Übergabeparameter lediglich das Tabellenhandle, um die interne Tabelle zu identifizieren, an die der Datensatz angehängt werden soll. Konnte Speicherplatz für den Datensatz reserviert werden, gibt die Funktion die Adresse der Speicherstelle zurück. Schlägt die Allokation des Speicherplatzes fehl, ist der Rückgabewert `NULL`.

Die Funktion `ItInsLine` fügt einen Datensatz an einer bestimmten Stelle ein. Die Schnittstelle der Funktion ist der nachfolgenden Tabelle zu entnehmen:

| Variable | Bedeutung |
|---|---|
| `itab` | Tabellenhandle zur Identifikation der Tabelle, in die der Datensatz einzufügen ist |
| `line` | Position, an der der Datensatz einzufügen ist |
| `void*` | Rückgabetyp der Funktion – Adresse des reservierten Speicherbereichs |

**Tabelle 4.12** Schnittstelle der Funktion ItInsLine

Unter Anwendung der Funktion `ItAppLine` wird die Funktion `AppendCustomer` des Moduls `CreateCustomerSAP` wie folgt realisiert:

```
RFC_RC AppendCustomer(RSCUSTOMEREXT   *pCustomerExt,
                      RFC_ERROR_INFO_EX  *pErrorInfo)
{
/* Prüfen, ob die Zeiger gültig sind */
   if(!pCustomerExt || !pErrorInfo)
      return RFC_FAILURE;
   void *pRow = NULL;
/* neue leere Zeile an die interne Tabelle anhängen */
   if(!(pRow = ItAppLine(m_TableParam[0].ithandle))){
      RfcLastErrorEx(pErrorInfo);
      return RFC_FAILURE;
   }
/* Daten in das SAP-Format übertragen
   und Daten an die durch pRow bezeichnete
   Speicherstelle kopieren */
   memcpy(pRow,
        &(ConvertDataFromExtToSAPFormat(pCustomerExt)),
        sizeof(_RSCUSTOMERSAP));
   return RFC_OK;
}
```

Die Realisierung der übrigen Funktionen wird nicht mehr explizit dargestellt. Sie erfolgt analog und kann dem Programm `CreateCustomerRFC` auf der Webseite zum Buch entnommen werden.

Das Ändern und Löschen von Datensätzen erfolgt durch die Funktionen `ItGupLine` und `ItDelLine` der RFC-API. Die Schnittstelle der Funktion `ItGupLine` lautet:

| Variable | Bedeutung |
|---|---|
| itab | Tabellenhandle zur Identifikation der Tabelle, in der der Datensatz steht |
| line | Index der Position, an der der Datensatz steht |
| void* | Rückgabetyp der Funktion; Adresse der Speicherstelle, an der der Datensatz steht |

**Tabelle 4.13** Schnittstelle der Funktion ItGupLine

Die Funktion `ItGupLine` gibt die Adresse der Speicherstelle des Datensatzes zurück. Der Inhalt der Speicherstelle kann somit durch das rufende Programm verändert werden.

Die Funktion zum Löschen von Datensätzen in einer internen Tabelle ist `ItDelLine`. Ihre Schnittstelle lautet:

| Variable | Bedeutung |
|---|---|
| itab | Tabellenhandle zur Identifikation der Tabelle, in der der zu löschende Datensatz steht |
| line | Index der Position, an der der Datensatz steht |
| int | Rückgabetyp der Funktion, durch dessen Wert signalisiert wird, ob die Operation erfolgreich ausgeführt wurde. Es gilt:<br>Rückgabewert = 0: erfolgreiche Ausführung<br>Rückgabewert > 0: Datensatz existiert nicht<br>Rückgabewert < 0: unbekannter Fehler |

**Tabelle 4.14** Schnittstelle der Funktion ItDelLine

Beim Löschen eines Datensatzes rücken alle nachfolgenden Datensätze um eine Position nach vorn. Besteht eine interne Tabelle aus drei Datensätzen und wird der zweite gelöscht, wird aus dem dritten Datensatz der neue zweite.

Der lesende Zugriff auf einen Datensatz erfolgt durch die Funktion `ItGetLine`. Der Aufbau ihrer Schnittstelle lautet:

| Variable | Bedeutung |
|---|---|
| itab | Tabellenhandle zur Identifikation der Tabelle, in der der zu lesende Datensatz steht |
| line | Index der Position, an der der Datensatz steht |
| void* | Rückgabetyp der Funktion – in ihm steht die Speicheradresse des Datensatzes |

**Tabelle 4.15** Schnittstelle der Funktion ItGetLine

Die Funktion `ItGetLine` soll gemäß der Dokumentation von SAP nur einen lesenden Zugriff auf den Datensatz der internen Tabelle ermöglichen. Eine Modifikation des Speicherinhalts der zurückgegebenen Speicherstelle soll nicht auf die interne Tabelle zurückwirken. Dies ist jedoch nicht der Fall. Die Funktion `ItGetLine` gibt ebenso wie die Funktion `ItGupLine` die Speicheradresse zurück, an der sich der Originaldatensatz in der internen Tabelle befindet. Damit wirkt eine Veränderung des Inhalts der Speicherstelle auch auf den Inhalt der internen Tabelle zurück. Um richtig zu arbeiten, müsste die Funktion `ItGetLine` den Inhalt der Speicherstelle in einen neu angeforderten Speicherbereich kopieren und die Adresse des neuen Speicherbereichs zurückgeben.

Folgende Funktion wird benutzt, um den Datensatz in einer internen Tabelle in einen anderen Speicherbereich zu kopieren:

```
int SAP_API ItCpyLine  (ITAB_H itab, unsigned line, void *dest)
```

Dabei wird der zu kopierende Datensatz über das Feld `line` identifiziert. Das Ziel des Kopiervorgangs wird im Feld `dest` angegeben. Das Tabellenhandle `itab` legt die anzusprechende interne Tabelle fest.

Das Gegenstück zur Funktion `ItCpyLine` ist folgende der RFC-API:

```
int SAP_API ItPutLine (ITAB_H itab, unsigned line, void* src)
```

Die Funktion kopiert den Inhalt des durch `src` identifizierten Speicherbereichs auf den durch `line` angesprochenen Datensatz. Die Tabelle wird wieder durch das Tabellenhandle identifiziert.

Die Anzahl der Datensätze in einer internen Tabelle wird mit dieser Funktion ermittelt:

```
unsigned SAP_API ItFill (ITAB_H itab)
```

Sie erhält als Argument das Handle der Tabelle. Die Funktion gibt die Anzahl der Datensätze in der durch das Handle identifizierten Tabelle zurück.

Mit folgender Funktion lässt sich ermitteln, wie viel Speicher für jeden Datensatz in der internen Tabelle angefordert wird:

```
unsigned SAP_API ItLeng (ITAB_H itab)
```

Ihr wird das Tabellenhandle zur Identifikation der internen Tabelle übergeben, und die Funktion gibt den Speicherbedarf zurück.

### 4.3.4 Übersicht über die Funktionen für interne Tabellen

Die nachfolgende Tabelle zeigt nochmals die Funktionen zum Umgang mit internen Tabellen in alphabetischer Übersicht.

| Funktion | Bedeutung |
|---|---|
| `ItAppLine` | Anhängen einer Zeile an eine interne Tabelle |
| `ItCpyLine` | Kopieren des Inhalts eines Datensatzes von einer internen Tabellen an einen durch `dest` angegebenen Speicherbereich |
| `ItCreate` | Anlegen einer internen Tabelle |
| `ItDelete` | Löschen einer internen Tabelle inklusive der Verwaltungsinformationen |
| `ItDelLine` | Löschen eines Datensatzes aus der internen Tabelle |
| `ItFill` | Anzahl der Datensätze in der internen Tabelle |
| `ItFree` | Löschen aller Datensätze aus der internen Tabelle. Die Verwaltungsinformationen bleiben erhalten. |
| `ItGetLine` | Rückgabe der Speicheradresse eines Datensatzes in der internen Tabelle |
| `ItGupLine` | Rückgabe der Speicheradresse eines Datensatzes in der internen Tabelle |
| `ItInsLine` | Einfügen eines Datensatzes in die interne Tabelle an der durch den Index angegebenen Position |
| `ItLeng` | Bytegröße eines zu speichernden Datensatzes |
| `ItPutLine` | kopiert den Inhalt der Speicherstelle `src` auf den durch Index identifizierten Datensatz |

**Tabelle 4.16** Funktionen zum Umgang mit internen Tabellen

## 4.4 Die Nachrichtenschleife

Die Nachrichtenschleife ist ein elementarer Bestandteil jedes externen Servers. In diesem Abschnitt werden wir uns noch einmal mit den Möglichkeiten ihrer Programmierung beschäftigen.

Bei unserem ersten Server in Kapitel 3 wurde schon darauf hingewiesen, dass zwei Arten der Programmierung einer Nachrichtenschleife möglich sind:

1. Blocking-Modus
2. Polling-Modus

Der Blocking-Modus wurde im vorherigen Kapitel erläutert. Hier wird die Programmierung einer Nachrichtenschleife im Polling-Modus vorgestellt. Für die Programmierung einer Nachrichtenschleife in diesem Modus bietet die RFC-API zwei Funktionen an:

- `RfcListen`
- `RfcWaitForRequest`

Der Unterschied zwischen beiden Funktionen besteht darin, dass die Funktion `RfcListen` nur prüft, ob eine Nachricht vom SAP-System vorliegt, und mit dem Ergebnis der Überprüfung sofort zurückkehrt. Dabei sind drei Rückgabewerte möglich, die die folgende Bedeutung haben:

| Rückgabewerte | Bedeutung |
|---|---|
| `RFC_OK` | Eine Nachricht von einem SAP-System liegt vor. |
| `RFC_RETRY` | Es liegt keine Nachricht vor. |
| `RFC_FAILURE` | Fehler in der Kommunikation SAP-System/RFC-Server. |

**Tabelle 4.17** Mögliche Rückgabewerte der Funktion RfcListen

Der Nachteil der Funktion `RfcListen` besteht darin, dass sie – aufgrund der sofortigen Rückkehr – permanent überprüfen muss, ob eine Nachricht vom SAP-System vorliegt. Der ausführende Prozess muss also ständig lauschen und wird nicht suspendiert, wodurch die Systembelastung ansteigt. Wenn Sie sich hiervon überzeugen möchten, sehen Sie sich beispielsweise unter Windows im Taskmanager die CPU-Nutzung eines Programms an, das eine Nachrichtenschleife mit der Funktion `RfcListen` realisiert.

Besser ist die Verwendung der Funktion `RfcWaitForRequest`. Die Schnittstelle der Funktion `RfcWaitForRequest` lautet:

| Variable | Bedeutung |
|---|---|
| `handle` | Verbindungshandle |
| `wtime` | Zeitintervall, in dem auf eingehende Nachrichten vom SAP-System gewartet werden soll |

**Tabelle 4.18** Schnittstelle der Funktion RfcWaitForRequest

Der Funktion wird somit zusätzlich das Zeitintervall übergeben, in dem sie auf Nachrichten vom R/3-System wartet. Der Thread wird für die Dauer des Zeitintervalls suspendiert. Er erhält somit keine CPU-Zeit zugeteilt, wodurch die Belastung des Systems enorm zurückgeht. Das Zeitintervall wird dabei in Sekunden angegeben. Die Rückgabewerte der Funktion `RfcWaitForRequest` sind identisch mit denen der Funktion `RfcListen`.

Sehen wir uns an, wie die Programmierung einer Nachrichtenschleife im Polling-Modus aussieht:

```
do{
   rc = RfcWaitForRequest(hConnection, 1);
   switch(rc){
      case RFC_RETRY:{
         rc = RFC_OK;
         break;
         }
      case RFC_OK:{
         if((rc = RfcDispatch(hConnection)) != RFC_OK){
            RfcLastErrorEx(&ErrorInfo);
            printf("%s\n", ErrorInfo.message);
         }
         break;
      }
   case RFC_FAILURE:{
      RfcLastErrorEx(&ErrorInfo);
      printf("%s\n", ErrorInfo.message);
      break;
   }
}while(rc == RFC_OK);
```

**Listing 4.10** Nachrichtenschleife im Polling-Modus

Eine Nachrichtenschleife im Polling-Modus besteht im Wesentlichen aus einer `DO-WHILE`-Schleife Die `DO-WHILE`-Schleife wird durchlaufen, solange die Schleifenkontrollvariable `rc` den Wert `RFC_OK` besitzt. Innerhalb der Schleife wird die Funktion `RfcWaitForRequest` aufgerufen. Die Funktion `RfcWaitForRequest` prüft, ob eine Nachricht vom SAP-System vorliegt, und kehrt mit dem Ergebnis der Überprüfung zurück.

Das Ergebnis der Abfrage wird in einer `case`-Anweisung ausgewertet. Liegt eine Nachricht vor, wird die Funktion `RfcDispatch` aufgerufen, die ihrerseits die Callback-Funktion ansteuert. Es sei explizit betont, dass die Funktion `RfcWaitForRequest` nur überprüft, ob eine Nachricht vorliegt. Sie veranlasst nicht, dass die Callback-Funktion angesteuert wird. Dasselbe gilt auch für die Funktion `RfcListen`.

Ein häufig geäußertes Argument gegen das Suspendieren eines Prozesses kritisiert, dass Anfragen von SAP-Systemen nun nicht mehr sofort ausgeführt werden. Aber dieses Argument ist wenig stichhaltig, wenn Sie sich ansehen, wie gering die maximale Verzögerung der Ausführung einer Anfrage von einem SAP-System ist. In unserem Beispiel würde die Verzögerung nur eine Sekunde betragen. Dies ist

eine Zeitspanne, die in der Praxis für die meisten Einsatzzwecke vollkommen belanglos sein dürfte.

Sollte die Zeitspanne für die Suspendierung des Prozesses von einer Sekunde dennoch nicht akzeptabel sein, kann sie verringert werden, indem Sie die Funktion `RfcListen` in Kombination mit einer Betriebssystemfunktion für die Suspendierung des Prozesses verwenden. Unter Linux beispielsweise stehen die Funktionen `sleep` für die Suspendierung im Sekundenbereich und `nanosleep` für die Suspendierung auf Nanosekunden-Ebene zur Verfügung. Im Bereich der Windows-Programmierung wird hierfür sehr häufig die Funktion `SLEEP` verwendet. Der `SLEEP`-Funktion suspendiert einen Thread für n Millisekunden. Die Modifizierung der Nachrichtenschleife sieht so aus:

```
do{
   rc = RfcListen(hConnection);
   switch(rc){
      case RFC_RETRY:{
/* Prozess fuer 300 Millisekunden suspendieren */
         Sleep(300);
         rc = RFC_OK;
         break;
      }
:
}
```

**Listing 4.11** Suspendieren des Prozesses in einer Nachrichtenschleife, wenn keine Nachrichten des SAP-Systems vorliegen

Betrachtet man abschließend den Programmieraufwand, den eine Nachrichtenschleife im Polling-Modus verursacht, wird klar, dass er auch nicht wesentlich höher als bei einer Nachrichtenschleife im Blocking-Modus ist. Somit ist deutlich geworden, dass Nachrichtenschleifen im Polling- und Blocking-Modus gleichwertig sind.

## 4.5 Alternativen für das Anmelden an ein R/3-System

Nun sollen noch einmal verschiedene Möglichkeiten für die Anmeldung an das R/3-System gezeigt werden. Vorgestellt werden:

- Arbeiten mit einer Konfigurationsdatei
- Load Balancing
- Funktion `RfcOpen`

### 4.5.1 Arbeiten mit einer Konfigurationsdatei

Auch bei einem externen Client besteht die Option, mit der Konfigurationsdatei *saprfc.ini* zu arbeiten. Möchten Sie eine Konfigurationsdatei verwenden, ist in dem String mit den Verbindungsargumenten für die Funktion `RfcOpenEx` das folgende Argument aufzunehmen:

```
dest=<Verweis auf Eintrag in der Datei saprfc.ini>
```

Mit diesem Wert liest die Funktion `RfcOpenEx` die Daten für die Verbindung aus der Datei, dabei werden alle anderen systemspezifischen Logon-Daten im String mit den Verbindungsdaten ignoriert. Das nachfolgende Beispiel zeigt, wie die Datei *saprfc.ini* für die Anmeldung an einen bestimmten Applikationsserver aufgebaut sein muss. Anstelle der IP-Adresse eines Applikationsservers kann übrigens auch ein vollständiger SAP Router-String verwendet werden.

```
DEST=RFCRECHNER
TYPE=A
ASHOST=10.10.34.131
SYSNR=01
RFC_TRACE=0
ABAP_DEBUG=0
USE_SAPGUI=0
```

Für den Aufbau der Datei *saprfc.ini* existieren mehrere Optionen. Um sich über alle Optionen zu informieren, wird von SAP eine Musterdatei mit dem RFC-SDK ausgeliefert. Unter Windows befindet sich die Datei im Verzeichnis *\SAP\FrontEnd\SAPgui\rfcsdk\text*.

Die Verbindungsargumente für die Verwendung der Datei *saprfc.ini* werden der Funktion `RfcOpenEx` wie folgt übergeben:

```
char ConParam[] = "DEST=RFCRECHNER CLIENT=099 "\
                  "USER=<User> PASSWD=<Password> "\
                  "LANG=DE ABAP_DEBUG=0
memset(&ErrorInfoEx, NULL,
       sizeof(RFC_ERROR_INFO_EX_EX));
if((hConnection = RfcOpenEx(ConParam, &ErrorInfoEx))
               == RFC_HANDLE_NULL){

   printf("%s\n", ErrorInfo.message);
   rc = RFC_FAILURE;
}
return rc;
```

### 4.5.2 Arbeiten mit Load Balancing

Für die Nutzung des Load Balancings gibt es zwei Alternativen:

- Verwendung einer Konfigurationsdatei
- direkte Angabe der IP-Adresse des Rechners, auf dem sich der Message-Server befindet

Die Konfigurationsdatei heißt *SAPMSG.INI*. In der Datei ist der Name des Rechners hinterlegt, auf dem sich der Message-Server des SAP-Systems befindet. Unter Windows befindet sich die Datei im Verzeichnis *C:\WINNT*. Die folgende Tabelle zeigt, wie die Verbindungsargumente bei der Nutzung von Load Balancing zu spezifizieren sind, wenn mit der Datei *SAPMSG.INI* gearbeitet wird:

| Argument | Bedeutung |
|---|---|
| `r3name` | Name des R/3-Systems. Zu dem Namen muss in der Datei *SAPMSG.INI* die Adresse des Rechners angegeben sein, auf dem sich der Message-Server befindet. |
| `group` | optionale Angabe der Logon-Gruppe |

**Tabelle 4.19** Aufbau der Verbindungsargumente für das Load Balancing unter Anwendung von SAPMSG.INI

Wird die Verbindung über einen SAP Router hergestellt, ist der Wert für das Argument `r3name` wie unten dargestellt aufzubauen:

```
r3name=<Name des SAP Routers><R/3-Systemname = Verweis auf
einen Eintrag in der Datei SAPMSG.INI>
```

Dabei verweist die Angabe für den R/3-Namen auf einen Eintrag in der Datei *SAPMSG.INI*, die sich auf dem SAP Router befindet.

Auch für die Angabe des SAP Routers bietet SAP die Option, mit einer Konfigurationsdatei zu arbeiten. Der Name der Datei lautet *SAPROUTE.INI*. Unter Windows befindet sich die Datei ebenfalls im Verzeichnis *C:\WINNT*. Wird mit zwei Konfigurationsdateien gearbeitet, ist der Wert für das Argument `r3name` folgendermaßen aufzubauen:

```
r3name=<Verweis auf SAP Router in der Datei SAPROUTE.INI>
<R/3-Systemname = Verweis auf Eintrag in der Datei SAPMSG.INI>
```

Wird ohne eine Konfigurationsdatei gearbeitet, müssen folgende Verbindungsargumente angegeben werden:

| Argument | Bedeutung |
|---|---|
| r3name | Name des R/3-Systems |
| mshost | Adresse des Rechners, auf dem sich der Message-Server befindet |
| group | optionale Angabe der Logon-Gruppe |

**Tabelle 4.20** Aufbau der Verbindungsargumente für das Load Balancing ohne Anwendung von SAPMSG.INI

Erfolgt die Herstellung der Verbindung über einen SAP Router, muss in dem Argument `mshost` zusätzlich der Name des SAP Routers vorangestellt werden.

### 4.5.3 Arbeiten mit der Funktion RfcOpen

In Kapitel 3 wurde die Funktion `RfcOpenEx` für die Anmeldung an das R/3-System verwendet. An dieser Stelle soll ihr Vorgänger `RfcOpen` vorgestellt werden. Die Funktion soll zwar in neuen Projekten nicht mehr genutzt werden, dennoch wird sie noch von SAP ausgeliefert.

Der Vorteil der Funktion `RfcOpen` liegt darin, dass die Verbindungsdaten in Strukturen hinterlegt werden, sodass anhand der Namen der Strukturfelder sofort ersichtlich ist, um was für eine Art von Information es sich handelt. Zudem wird in Dialogfenstern mit Eingabefeldern gearbeitet. Eine Struktur bietet hier den Vorteil, das Eingabefeld direkt mit dem Strukturfeld zu verknüpfen, was bei einem String nicht so ohne weiteres möglich ist.

Die Funktion `RfcOpen` verwendet für die Spezifizierung der Verbindungsdaten zwei Strukturen. In der ersten Struktur werden die benutzerabhängigen Verbindungsdaten gespeichert und in der zweiten die systemabhängigen.

Die Struktur für die benutzerabhängigen Verbindungsdaten ist `RFC_OPTIONS`. Sie verfügt über folgende Felder:

| Felder | Bedeutung |
|---|---|
| destination | Verweis auf Eintrag in der Datei *saprfc.ini* |
| mode | Art der Verbindung |
| connopt | Verweis auf Struktur mit den Systemdaten für die Verbindung |
| client | Client |
| user | Anwender |
| password | Passwort des Anwenders |

**Tabelle 4.21** Aufbau der Struktur RFC_OPTIONS

| Felder | Bedeutung |
|---|---|
| language | Anmeldesprache |
| trace | Trace aktivieren |

**Tabelle 4.21** Aufbau der Struktur RFC_OPTIONS (Forts.)

In den Feldern client, user, password und language werden die anwenderspezifischen Logon-Daten hinterlegt. Durch das Feld trace wird die Protokollierung der aufgerufenen Funktionen und gesendeten Daten in einer Trace-Datei aktiviert.

Über das Feld mode wird gesteuert, auf welche Strukturvariable im Feld connopt zu verweisen ist. Die Strukturvariable enthält die Systemdaten. In der Praxis werden zwei Modi häufig verwendet:

| Modus | Zusätzliche Struktur |
|---|---|
| RFC_MODE_R3ONLY | RFC_CONNOPT_R3ONLY |
| RFC_MODE_VERSION_3 | RFC_CONNOPT_VERSION_3 |

**Tabelle 4.22** Häufig verwendete Modi für die Struktur RFC_OPTIONS

In diesem Abschnitt wird nur auf die Option RFC_MODE_VERSION_3 eingegangen, da sie das Load Balancing unterstützt. Bei der Option RFC_MODE_VERSION_3 werden die Systemdaten in einer Strukturvariable des Typs RFC_CONNOPT_VERSION_3 übergeben. Der Aufbau der Struktur RFC_CONNOPT_VERSION_3 ist folgender:

| Feld | Bedeutung |
|---|---|
| hostname | IP-Adresse des Applikationsservers, falls die Anmeldung an einem bestimmten Applikationsserver erfolgen soll |
| sysnr | Systemnummer |
| use_load_balancing | Kennzeichen zur Festlegung, ob für die Anmeldung das Load Balancing verwendet werden soll. Ist der Wert ungleich null, wird das Load Balancing verwendet. |
| lb_host | IP-Adresse des Applikationsservers, auf dem der Message-Server läuft |
| lb_system_name | Systemnummer |
| lb_group | Logon-Gruppe, an die man sich anmelden möchte |
| use_sapgui | Kennzeichen, ob das SAP GUI aktiv sein soll |

**Tabelle 4.23** Felder der Struktur RFC_CONNOPT_VERSION_3

Das Feld `use_load_balancing` der Struktur `RFC_CONNOPT_VERSION_3` entscheidet darüber, ob die Load-Balancing-Option genutzt wird oder nicht. Wird Load Balancing eingesetzt, ist im Feld `lb_host` die IP-Adresse des Applikationsservers anzugeben, auf dem der Message-Server läuft. Sollte das verteilte R/3-System zwar aus mehreren Applikationsservern bestehen, diese aber nicht zu Logon-Gruppen zusammengefasst sein, ist die Default-Logon-Gruppe `NULL` im Feld `lb_group` anzugeben. Ferner werden Angaben in den Feldern `hostname` und `sysnr` nicht berücksichtigt, sobald die Option `use_load_balancing` gesetzt wurde.

# Index

## S