MIDRANGE
HANDBÜCHER

3 Kapitel

Klassische Anwendungs-Entwicklung

Walter Becker, Ronald Böhmer, Birgitta Hauser, Klaus-Peter Luttkus,
Carmen Rasche, Rainer Ross, Jörg Zeig

IBM POWER Systems i Softwareentwicklung und -modernisierung

Handbuch für Strategien und Leitfaden für moderne Programmierung
sowie Tipps & Tricks

ITP VERLAG

Klassische Anwendungs-Entwicklung

3.1 ILE-Konzepte
3.1.1 Was verbirgt sich hinter dem Mysterium ILE?
3.1.2 Warum sollte man ILE-Konzepte verwenden?
3.1.3 Anwendungsmodernisierung mit ILE-Konzepten
3.1.4 Programmierkonzepte
3.1.4.1 Original Program Model (OPM)
3.1.4.2 Integrated Language Environment (ILE)
3.1.4.3 Objektorientierte Programmierung (OOP)
3.1.4.4 Vergleich OPM, ILE und OOP
3.1.5 Quellenstruktur
3.1.5.1 Main Procedure
3.1.5.2 (Sub-)Procedures
3.1.5.2.1 Interne Prozeduren
3.1.5.2.3 Funktionen
3.1.5.2.4 Globale und lokale Definitionen
3.1.6 Prototyping-Basics
3.1.6.1 Procedure Interface – Eingangsparameter
3.1.6.1.1 Input-Only-Parameter
3.1.6.1.2 Parameteroptionen
3.1.6.2 Prototyp
3.1.6.2.1 Prototypen für Programme und Prozeduren
3.1.7 ILE-Objekte
3.1.7.1 Module – Basis für Programme und Serviceprogramme
3.1.7.1.1 CL-Befehl DSPMOD – Modulinformationen anzeigen
3.1.7.2 Programme
3.1.7.2.1 Erstellen von ILE-Programmen
3.1.7.2.1.1 Bind by Copy – Module in Programme einbinden
3.1.7.2.1.2 CRTPGM – Programm erstellen
3.1.7.2.1.3 CRTBNDXXX – Binder-Programm erstellen
3.1.7.2.2 DSPPGM – Programm anzeigen
3.1.7.3 Verwendung von Modulen = modulare Programmierung?
3.1.7.3.1 Von der Copy-Strecke zum Modul
3.1.7.3.1.1 Ursprünglicher Source Code mit Copy-Strecken
3.1.7.3.1.2 Modifizierter Source Code: Modul statt Copy-Strecke

3.1.7.3.2 Von der Main-Procedure zur Sub-Procedure
3.1.7.3.3 Von der Sub-Procedure zur Funktion
3.1.7.4 Serviceprogramm
3.1.7.4.1 Erstellen von Serviceprogrammen
3.1.7.4.2 Bind by Reference über die Signatur des Serviceprogramms
3.1.7.4.3 CRTSRVPGM – Serviceprogramm erstellen
3.1.7.4.4 DSPSRVPGM – Serviceprogramm anzeigen
3.1.7.4.5 Vom Modul zum Serviceprogramm
3.1.7.5 Neue Prozeduren in Serviceprogrammen
3.1.8 Bindersprache
3.1.8.1 Erstellen/Aktualisieren eines Serviceprogramms mit Bindersprache
3.1.8.2 Komponenten der Bindersprache
3.1.8.2.1 Befehl STRPGMEXP – Programmexportliste starten
3.1.8.2.2 Bindersprache: Befehl EXPORT
3.1.8.2.3 Bindersprache: Befehl ENDPGMEXP
3.1.8.2.4 Beispiel Binderquelle
3.1.8.3 CL-Befehl: RTVBNDSRC – Binderquelle abrufen
3.1.8.4 Beispiel: Serviceprogramm mit fixer Signatur
3.1.8.5 Beispiel: Serviceprogramm mit mehreren Signaturen
3.1.9 Binderverzeichnis
3.1.9.1 Binderverzeichnis erstellen
3.1.9.2 Binder-Verzeichniseinträge erstellen
3.1.9.3 Regeln zur Verwendung von Binderverzeichnissen
3.1.9.4 Zweistufige Kompilierung mit Binderverzeichnissen
3.1.9.5 Einstufige Kompilierung mit Binderverzeichnissen
3.1.9.6 Befehl CRTBNDxxx mit Binderverzeichnis
3.1.9.6.1 H-Bestimmungen für ILE-Optionen in RPG-Quellen
3.1.9.6.1.1 Compiler-Direktiven für ein- oder zweistufige Kompilierung
3.1.9.6.2 CL-Befehl DCLPRCOPT für ILE-Optionen in CL-Quellen
3.1.10 Ausführung von Programmen und Serviceprogrammen
3.1.10.1 Aktivierung von Programmen und Serviceprogrammen
3.1.10.2 Deaktivierung von Programmen und Serviceprogrammen
3.1.10.2.1 Deaktivierung von OPM-Programmen
3.1.10.2.2 Deaktivierung von ILE-Programmen

3.1.10.3 Was sind Aktivierungsgruppen?
3.1.10.4 Standard/Default-Aktivierungsgruppen
3.1.10.4.1 Programme und Service-Programme in Default-Aktivierungsgruppen
3.1.10.4.1.1 OPM-Programme in der Default-Activation-Group
3.1.10.4.1.2 ILE-Programme in der Default-Activation-Group
3.1.10.4.1.3 Serviceprogramme in der Default-Activation-Group
3.1.10.5 Aktivierungsgruppe *NEW
3.1.10.6 Aktivierungsgruppe *CALLER
3.1.10.7 Benannte Aktivierungsgruppen
3.1.10.7.1 Programme in benannten Aktivierungsgruppen
3.1.10.7.2 Serviceprogramme in benannten Aktivierungsgruppen
3.1.10.7.3 Aktivierungszeitpunkt von Serviceprogrammen
3.1.10.8 Aktivierungsgruppen und Dateiüberschreibungen
3.1.10.8.1 Dateiüberschreibungen setzen
3.1.10.8.2 Dateiüberschreibungen zurücknehmen
3.1.10.9 Aktivierungsgruppen und Commit-Definitionsbereich
3.1.10.9.1 Commitment-Steuerung und SQL
3.1.10.10 Aktivierungsgruppen und Open Data Paths (ODPs)
3.1.10.10.1 Empfehlungen im Umgang mit Aktivierungsgruppen
3.1.11 Tipps für ILE-Anwendungen

3.2 RPG **Seite 5**
3.2.1 RPG-Versionen
3.2.2 RPG-Formate
3.2.2.1 Fixformat RPG
3.2.2.2 Free-Format RPG (erste Version)
3.2.2.3 Free-Format RPG (finale Version)
3.2.3 RPG-Editoren
3.2.4 RPG-Bestimmungen
3.2.4.1 Kommentare
3.2.4.2 Steuerbestimmungen
3.2.4.3 RPG-Dateibestimmungen
3.2.4.4. Definitionsbestimmungen
3.2.4.4.1 Schlüsselwörter für Definitionsbestimmungen
3.2.4.5 RPG-Rechenbestimmungen
3.2.4.5.1 Groß- und Kleinschreibung
3.2.4.5.2 Wertzuweisungen
3.2.5 RPG und Embedded SQL
3.2.5.1 Welche Vorteile bietet der Einsatz von embedded SQL?
3.2.5.2 Embedded SQL und Query-Optimierung
3.2.5.3 Voraussetzungen für embedded SQL
3.2.5.3.1 Quellen-Teildateien mit embedded SQL
3.2.5.3.2 Codierung von SQL-Befehlen im Quellcode
3.2.5.3.2.1 Codierung von SQL-Befehlen im fixen RPG-Format
3.2.5.3.2.2 Codierung von SQL-Befehlen in Free-Format RPG
3.2.5.3.2.3 Codierung von SQL-Befehlen in Cobol
3.2.5.3.2.4 Kommentare in (embedded) SQL-Befehlen
3.2.5.3.3 Erstellung von Objekten mit embedded SQL
3.2.5.3.3.1 Kompilierungsbefehle
3.2.5.3.3.2 Kompilierung von Modulen/Programmen mit embedded SQL
3.2.5.3.3.3 SQL Communication Area (SQLCA)
3.2.5.3.3.4 SQL-Befehl SET OPTION
3.2.5.4 Verarbeitungsmethoden
3.2.5.5 Verwendung von Variablen
3.2.5.5.1 Host-Variablen
3.2.5.5.2 Host-Strukturen
3.2.5.5.3 SQL-Datentypen und RPG-Äquivalent

3.2.5.5.4 Indikator-Variablen
3.2.5.5.5 Parameter-Marker
3.2.5.5.6 Namenskonventionen
3.2.5.6 Fehlerbehandlung in embedded SQL
3.2.5.6.1 Fehlerbehandlung über SQLCODE
3.2.5.6.2 Fehlerbehandlung über SQLSTATE
3.2.5.6.3 IBM Information Center – SQLSTATEs und SQLCODEs
3.2.5.7 Statisches SQL ohne Cursor
3.2.5.7.1 SQL-Befehl SET
3.2.5.7.1.1 Verwendung von SPECIAL REGISTERs
3.2.5.7.1.2 Berechnung von Zeitdifferenzen
3.2.5.7.1.3 Verwendung von SQL-Prädikaten
3.2.5.7.1.4 Verwendung von SQL-skalaren Funktionen
3.2.5.7.2 Massendatenverarbeitung
3.2.5.7.3 Einzelsatz-Verarbeitung
3.2.5.7.3.1 SQL-Befehl SET in Verbindung mit SELECT-Anweisung
3.2.5.7.3.2 SQL-Befehl SELECT ... INTO
3.2.5.7.3.3 SQL-Befehl VALUES ... INTO
3.2.5.8 Dynamisches SQL ohne Cursor
3.2.5.8.1 SQL-Befehl EXECUTE IMMEDIATE
3.2.5.8.2 SQL-Befehle PREPARE und EXECUTE
3.2.5.8.1.1 SQL-Befehl PREPARE
3.2.5.8.1.2 SQL-Befehl EXECUTE
3.2.5.8.1.3 Dynamisches SQL mit PREPARE und EXECUTE – ein Beispiel
3.2.5.9 SQL-Befehl VALUES ... INTO in dynamischem SQL
3.2.5.10 Definition und Verwendung der Cursors
3.2.5.10.1 SQL-Befehl DECLARE CURSOR
3.2.5.10.1.1 SQL-Befehl DECLARE CURSOR für serielle Cursor
3.2.5.10.1.2 SQL-Befehl DECLARE CURSOR für Scroll („rollende") Cursor
3.2.5.10.1.3 SQL-Befehl DECLARE CUSROR und Commitment-Steuerung
3.2.5.10.1.4 SQL-Befehl SELECT in Verbindung mit Cursor-Deklarationen
3.2.5.11 SQL-Befehl OPEN Cursor

3.2.5.12 SQL-Befehl FETCH – Zeilen/Datensätze einlesen
3.2.5.12.1 SQL-Befehl FETCH mit seriellem Cursor
3.2.5.12.2 SQL-Befehl FETCH mit Scroll Cursor
3.2.5.13 SQL-Befehl CLOSE Cursor
3.2.5.14 Statisches SQL mit Cursor-Verarbeitung – ein Beispiel
3.2.5.15 Fortschreiben von Datensätzen bei Cursor-Verarbeitung
3.2.5.16 Multiple Row Fetches – Einlesen von Datenblöcken
3.2.5.17 Dynamisches SQL mit Cursor
3.2.5.17.1 Dynamisches SQL mit fixer Auswahlliste
3.2.5.17.1.1 SQL-Befehl PREPARE mit dynamischen SELECT-Statements
3.2.5.17.1.2 SQL-Befehl DECLARE CURSOR für dynamisches SQL
3.2.5.17.1.3 Dynamisches SQL mit fixer Auswahlliste – Beispiel
3.2.5.17.1.4 Parameter-Marker – Verwendung von Variablen in dynamischem SQL
3.2.5.17.1.4.1 SQL-Befehl OPEN CURSOR in Verbindung mit Parameter-Markern
3.2.5.17.1.4.2 Parameter-Marker in dynamischer Cursor-Verarbeitung – Beispiel
3.2.5.17.1.5 Dynamisches SQL mit fixer Auswahlliste oder statisches SQL?
3.2.5.17.2 Dynamisches SQL mit variabler Auswahlliste
3.2.5.17.2.1 SQL Descriptor Area (SQLDA)
3.2.5.17.2.2 SQL-Befehl DESCRIBE – Zuweisung der SQL Descriptor Area
3.2.5.17.2.3 SQL-Befehl PREPARE in Verbindung mit SQL Descriptor Area
3.2.5.17.2.4 SQL-Befehl FETCH in Verbindung mit SQL Descriptor Area
3.2.5.17.2.5 Dynamisches SQL mit flexibler Auswahlliste – Beispiel
3.2.5.18 Verarbeitung von Result Sets aus Stored Procedures
3.2.5.18.1 Host-Variablen mit Datentyp RESULT_SET_LOCATOR
3.2.5.18.2 SQL-Befehl CALL – Aufruf von Stored Procedures
3.2.5.18.3 SQL-Befehl ASSOCIATE LOCATORS – Zuordnen von Result-Sets
3.2.5.18.4 SQL-Befehl ALLOCATE CURSOR – Result-Set öffnen
3.2.5.18.5 SQL-Befehl FETCH – Verarbeiten von Result-Sets aus Procedures

3.2.5.18.6	SQL-Befehl CLOSE CURSOR für Result-Set
3.2.5.18.7	Verarbeiten von Result-Sets aus Stored Procedures – Beispiel
3.2.5.19	Diagnostics Area – SQL-Befehl GET DIAGNOSTICS
3.2.5.19.1	Ermitteln von Statement- und Connection-Informationen
3.2.5.19.2	Ermitteln von Bedingungsinformationen
3.2.6	RPG und Daten
3.2.6.1	Daten einer Datei mit RPG lesen
3.2.6.1.1	READ – Lesen eines Datensatzes
3.2.6.1.2	READE – Lesen eines Datensatzes
3.2.6.1.3	CHAIN – Positionieren und Lesen eines Datensatzes
3.2.6.2	Schlüsseldefinitionen in RPG
3.2.6.2.1	KLIST/kfld
3.2.6.2.2	Schlüsseldatenstrukturen
3.2.7	Datenmanipulation mit RPG
3.2.7.1	Hinzufügen von neuen Datensätzen mit RPG nativ
3.2.7.2	Hinzufügen von neuen Datensätzen mit RPG und embedded SQL
3.2.7.3	Löschen bestehender Datensätze mit RPG nativ
3.2.7.4	Löschen bestehender Datensätze mit RPG und embedded SQL
3.2.7.5	Verändern bestehender Datensätze mit RPG nativ
3.2.7.6	Verändern bestehender Datensätze mit embedded SQL
3.2.8	Built-in Functions
3.2.8.1	Umwandlung vom dezimalen Zeichenformat in Character
3.2.8.2	Umwandlung von Zeichenfeldern in gepackte dezimale Daten
3.2.8.3	Teilzeichenfolgen (Substrings)
3.2.8.4	Ersetzen von Zeichenfolgen
3.2.8.5	Suchen und Ersetzen von Zeichenfolgen
3.2.8.6	Verknüpfen von Zeichenketten und Entfernen von Leerstellen

3.2.8.7 Umwandlung von Zeichen in Integer-Format Seite 9
3.2.8.8 Ermitteln von Größe, Länge und Dezimalstellen
3.2.8.9 Anzahl der Elemente
3.2.8.10 Suchfunktionen
3.2.8.11 Zeichenfolgen umsetzen
3.2.8.12 Kombinieren von Built-in Functions
3.2.8.13 Datums- und Zeitoperanten
3.2.8.14 Rechnen mit Datum und Zeit

3
Seite 10

3.3 COBOL
3.3.1 CL – Control Language (Steuersprache)
3.3.2 CL – Neuerungen V5R2
3.3.3 CL – Neuerungen V5R3
3.3.3.1 Neue Typen von Variablen
3.3.3.2 Deklaration und Verarbeitung von Dateien
3.3.3.3 Strukturbefehle im CL
3.3.3.4 Die neuen CL-Befehle in einem Musterprogramm
3.3.3.5 Informationen aus der Umwandlungsliste.
3.3.4 CL – Neuerungen V5R4
3.3.4.1 Erweiterungen in der DCL-Anweisung
3.3.4.2 Definition von Datenstrukturen
3.3.4.3 Pointer Variablen und Pointer-Arithmetik im CL
3.3.4.4 Subroutinen im CL
3.3.4.5 Declare Processing Options
3.3.4.6 Abschluss
3.3.5 CL – Neuerungen in V6R1
3.3.5.1 INCLUDE
3.3.5.2 DCLPRCOPT
3.3.5.3 CLOSE
3.4 API – Begriffsdefinition
3.4.1 API – Suchen und Finden
3.4.2 Kommunikation mit APIs
3.4.3 Die Bibliothek QSYSINC
3.4.4 Der Aufruf von APIs
3.4.5 Datentypen bei der Verwendung von APIs
3.4.6 Der optionale Parameter „Error-Code"
3.4.7 API QUSLFLD
3.4.7.1 APIs für User Spaces
3.4.7.2 Gemeinsame Datenstrukturen von List APIs im User Space

3.1 ILE-Konzepte

Gleichzeitig mit RPG IV wurden auch die ILE-Konzepte eingeführt, nach denen in und mit RPG, aber auch mit anderen Programmiersprachen, wie zum Beispiel Cobol, CL oder C, programmiert werden kann. Aus diesem Grund werden zum Beispiel RPG IV und ILERPG im Sprachgebrauch oft synonym verwendet, was jedoch von der Sache her nicht korrekt ist.

Viele RPG-Programmierer schreiben ihre Programme „nur" noch in modernem RPG IV. Jedoch auch der Einsatz der letzten RPG-Erweiterungen – etwa qualifizierte Datenstrukturen, Built-in-Funktionen, Free-Format-Codierung oder XML-Verarbeitung –, bedeutet noch nicht, dass nach und mit ILE-Konzepten programmiert wird.

3.1 Seite 2

3.1.1 Was verbirgt sich hinter dem Mysterium ILE?

ILE steht für Integrated Language Environment und war ursprünglich dazu gedacht, Prozeduren, Funktionen und Methoden, die in anderen Programmiersprachen geschrieben wurden, aus den ILE-Sprachen (zum Beispiel RPG, CL oder Cobol) heraus aufrufen zu können.

Während im klassischen RPG sowie in Cobol und CL der Source Code in Programme umgewandelt wird, die über entsprechende CALL-Befehle aufgerufen und ausgeführt werden, gibt es in anderen Sprachen noch weitere ausführbare Konstrukte beziehungsweise Prozeduren und Funktionen. In RPG III gab es keine Möglichkeit, solche Konstrukte zu verwenden. Mit Hilfe der ILE-Konzepte ist es jedoch möglich, solche Prozeduren und Funktionen direkt aus den ILE-Sprachen heraus aufzurufen und auszuführen.

Die ILE-Konzepte erlauben jedoch nicht nur, Prozeduren, Funktionen und Methoden auszuführen, die in anderen Programmiersprachen geschrieben wurden, sondern auch, mit den ILE-Sprachen solche Konstrukte zu generieren. Zudem erlauben es die ILE-Konzepte, Programme aus mehreren Programmteilen zu bilden, die ihrerseits in unterschiedlichen Programmiersprachen verfasst wurden. Der Grundgedanke von ILE ist: Verwende die Programmiersprache, die für die aktuelle Aufgabe am besten geeignet ist.

Ergänzung 29./2014

3.1.1 Seite 2

3.1.2 Warum sollte man ILE-Konzepte verwenden?

Wir beherrschen eh' nur eine Programmiersprache und Programme in RPG oder CL können über CALL-Befehle aufgerufen werden.

ILE-Konzepte erlauben eine hochmodulare Programmierung, in der wiederverwendbarer Code separiert wird und – wo immer er benötigt wird – integriert werden kann. Große Programme werden dazu in Einzelteile (Prozeduren) aufgesplittet, die wiederverwendbaren Code beinhalten.

Ja! Aber das machen wir doch heute auch, wir teilen Programme in Subroutinen auf, die mehrfach aufgerufen werden!

Was aber, wenn die gleiche Subroutine in mehreren Programmen verwendet werden muss?

Dann kopieren wir die Subroutine oder legen den Source Code in eine Copy-Strecke und integrieren diese, oder wir generieren ein Unterprogramm.

Wird Source Code kopiert, so muss im Falle einer Änderung diese in allen Duplikaten vollzogen werden, was zum einen mehrfachen Zeitaufwand bedeutet und es zum anderen schwierig macht, jeweils alle Duplikate zu finden.

Bei der Verwendung von Copy-Strecken muss der Source Code nur einmal gewartet werden. In den Copy-Strecken kann und muss auf globale Variablen außerhalb der Copy-Strecke zugegriffen werden. Setzt man Copy-Strecken ein, so muss sichergestellt werden, dass innerhalb der Copy-Strecken nicht versehentlich solche globalen Variablen verändert werden, denn das kann zu unerwarteten Ergebnissen führen.

Kapselt man die Subroutine in ein Programm, so muss der Source Code nur einmal gewartet werden, außerdem können Daten nur über Parameter ausgetauscht werden. Der dynamische Aufruf von Programmen erzeugt einen Overhead, das heißt, Speicher muss reserviert und initialisiert werden, Dateien müssen geöffnet werden etc. Da für den Source Code eines Programms jeweils eine eigene Teildatei verwendet wird, hält sich die Anzahl der Unterprogramme in Grenzen. So wird man selten Unterprogramme finden, die aus zehn oder noch weniger Statements bestehen.

3.1.2
Seite 2

Kapselt man stattdessen die Subroutine in eine Prozedur, kann man diese wie ein Programm über entsprechende CALL-Prozedur-Befehle aufrufen. Prozeduren sind kleine Blackboxen, in denen bestimmte Aufgaben ausgeführt werden. Der Austausch von Daten erfolgt über Parameter. Prozeduren können im Programmcode als interne Prozeduren hinterlegt und somit als „Programm im Programm" angesehen werden. Mehrere Prozeduren können in einer Teildatei (unabhängig) vom Programmcode zusammengefasst und einzeln aufgerufen werden. Die Teildateien können kompiliert und anschließend entweder in Programme oder Serviceprogramme eingebunden werden. Sofern die Prozeduren exportiert werden, können sie von außen, das heißt aus anderen Programmen und Prozeduren heraus aufgerufen werden, wo immer sie benötigt werden.

Durch das Binden der Module entfällt der Overhead, der beim Aufruf von Programmen entsteht. Prozeduren, auch wenn sie außerhalb des Programms codiert wurden, werden verarbeitet als wären sie Bestandteil des Programms.

Prozeduren können und sollten nach Funktionalität in Teildateien zusammengefasst werden. Ein Beispiel: Alle Prozeduren für die Datums- und Zeitrechnung werden in einer Teildatei gesammelt, in einer anderen Teildatei sind alle String-Handling-Prozeduren enthalten und in einer dritten Teildatei sind alle Prozeduren hinterlegt, die auf eine bestimmte Datei/Tabelle zugreifen.

Benötigt der Programmierer eine Datumsroutine, so schaut er in die Teildatei, in der die Datumsroutinen hinterlegt sind. Findet er die Routine, die er benötigt, verwendet er sie. Findet er sie nicht, fügt er sie hinzu, und der nächste Programmierer, der die gleiche Routine benötigt, kann sie ebenfalls verwenden.

Das Schöne an Prozeduren ist, dass sie einzeln codiert und direkt aufgerufen und getestet werden können; im Gegensatz zu Subroutinen, die nur innerhalb eines Programms oder einer Prozedur ausgeführt werden können.

Wenn man sich näher mit den ILE-Konzepten beschäftigt, so wird man feststellen, dass man die Programmierrichtung überdenken muss. Klassische Programme wurden – auch bedingt durch den RPG-Zyklus – Top-down, also von oben nach unten codiert. Mit ILE geht man in der umgekehrten Richtung vor, das heißt, zunächst plant und designt man die einzelnen Prozeduren, und erst zum Schluss setzt man sie zu einem Programm zusammen.

Das Schwierigste beim Einsatz der ILE-Konzepte ist die modulare Denkweise, das heißt, der klassische RPG- und Cobol-Programmierer muss zuerst lernen, **modular zu denken.** Das geht nicht von heute auf morgen, sondern ist ein fortlaufender Prozess. Aber mit der Zeit werden die Prozeduren immer überschaubarer.

In einer echten ILE-Umgebung wird es nur noch kleine Programme und sehr viele noch kleinere Prozeduren geben. Aus den Programmen und Prozeduren werden weitere Prozeduren aufgerufen und ausgeführt. Nur wenige Prozeduren werden mehr als 50 – 100 Codezeilen enthalten.

3.1.3 Anwendungsmodernisierung mit ILE-Konzepten

Mit Hilfe der ILE-Konzepte ist es möglich, selbst alte, gewachsene Software zu modernisieren. Das Programmiererteam sollte sich dazu zunächst zusammensetzen und festlegen, welche Prozeduren benötigt und daher erstellt werden sollten. Für den Einstieg fängt man mit kleinen allgemein wiederverwendbaren Prozeduren an, zum Beispiel mit Datumsroutinen und String-Handling-Funktionen.

Bei den anwendungsspezifischen Prozeduren beginnt man zunächst mit den Prozeduren für die Eingabeprüfungen. Beispiele dafür sind „StatusErfasst“, „StatusGültig“ etc. Was die sonstigen anwendungsindividuellen Prozeduren angeht, so nimmt man sich zuerst die Subroutinen vor, von denen man weiß, dass sie in mehreren Programmen codiert wurden. Die neuen Prozeduren sollten nach Funktionalität und nicht nach Programmzugehörigkeit gruppiert werden. So sollten alle Datumsroutinen in einem Modul oder Service-Programm zusammengruppiert werden. Alle String-Handling-Funktionen werden in einem anderen Modul oder Service-Programm konzentriert. Alle Zugriffe auf den Artikel-Stamm werden in einem weiteren Modul oder Service-Programm eingebunden … usw.

Prozedurnamen sind im Gegensatz zu Programmnamen nicht auf zehn Zeichen beschränkt. Deshalb können und sollten kryptische Namen wie zum Beispiel AY123FA, vermieden und stattdessen sprechende Namen verwendet werden. Da jedoch voraussichtlich eine Vielzahl von Prozeduren erstellt werden wird, sollte man vorab Namenskonventionen festlegen. Zum einen können dadurch doppelte Prozeduren vermieden werden, zum anderen können durch saubere Namenskonventionen vorhandene Prozeduren von allen Programmierern leichter lokalisiert werden.

Ergänzung 29./2014

3.1.3
Seite 2

Die Verwendung von sprechenden Prozedurnamen hat noch einen kleinen, jedoch nicht zu unterschätzenden Nebeneffekt. Man erreicht nämlich eine „Dokumentation durch Codierung", so dass selbst ein fremder Programmierer den Quelltext überblicken kann, auch wenn keine (weiteren) Kommentare im Quellcode vorhanden sind, wie das folgende Beispiel zeigt:

```
/Free
   ... sonstiger Source Code

   If genugBestandLager(XYART: XYMGE: XYLAG);
      ReserviereBestandLager(XYART: XYMGE: XYLAG);
   ElseIf genugBestand(XYART: XYMGE);
      ReserviereBestand(XYART: XYMGE: XYLAG);
   Else;
      SendeFehler('Nicht genug Bestand');
   EndIf;

   ... sonstiger Source Code
/EndIf
```

Dokumentation durch Codierung

Wiederverwendbare Prozeduren können designt, codiert, umgewandelt und getestet werden, ohne die ursprünglichen Programme anzufassen. Da es sich bei den wiederverwendbaren Prozeduren um „Miniprogramme" handelt, ist es möglich, immer mal wieder eine Prozedur so nebenher zu erstellen.

Sofern man die Anwendungsmodernisierung ernsthaft vorantreiben will, sollte man den Fokus zunächst darauf legen, Prozeduren, die häufig eingesetzt werden können zu planen, zu erstellen und zu testen. Im Anschluss daran können diese Prozeduren in dem vorhandenen Quell-Code integriert werden. Im Hinblick darauf sollte man auch die alten RPG-III-Programme mit Hilfe des CL-Befehls CVTRPGSRC (RPG-Quellen konvertieren) nach RPG IV portieren, zumindest dann, wenn man nicht umhin kommt, ein solch altes Programm zu modifizieren.

Wann immer ein nach RPG IV portiertes, „altes" Programm modifiziert werden muss, können in diesem Programm vorhandene Subroutinen oder auch einzelne Codezeilen durch einen Aufruf von neuen, getesteten Prozeduren ersetzt werden. Auf diese Weise wird der alte Source Code nach und nach durch Prozeduraufrufe ersetzt, was den ursprünglichen Quellcode in der Regel beträchtlich reduziert und wesentlich übersichtlicher macht.

Irgendwann in der Zukunft werden alle Programme modernisiert, strukturiert und wieder einfach zu warten sein, was mit Sicherheit besser ist, als irgendeine neue (Standard-)Software zu kaufen, die nur zum Teil den Anforderungen und vorhandenen Abläufen entspricht.

3.1.4 Programmierkonzepte

Man unterscheidet zwischen den folgenden Programmierkonzepten:

- Original Program Model (OPM)
- Integrated Language Environment (ILE)
- Object Oriented Programming (OOP)

3.1.4.1 Original Program Model (OPM)

Unter Original Program Model (OPM) versteht man die klassische, auch durch den RPG-Zyklus bedingte Top-Down-Programmierung. Bei diesem Modell werden nur monolithische Programme geschrieben. Innerhalb der Programme können nur globale Variablen definiert werden, das heißt, auf alle Variablen kann sowohl aus dem Hauptprogramm als auch aus jeder einzelnen Subroutine zugegriffen werden.

Eine Strukturierung des Source Codes in OPM-Programmen kann durch Subroutinen oder aber durch den Aufruf von weiteren (Unter-)Programmen erfolgen. Wird der gleiche Source Code in mehreren Programmen benötigt, so wird er in der Regel kopiert und gegebenenfalls modifiziert. Eine Auslagerung in Copy-Strecken oder in eigene Unterprogramme findet eher selten statt. Bei einer Änderung muss dann nämlich jede kopierte Version gefunden und geändert werden, was einen hohen Zeitaufwand erfordert und zudem fehleranfällig sein kann.

Klassische RPG- und Cobol-Programme sind in diesem Stil geschrieben. Viele dieser Programme wurden bereits vor Jahren geschrieben und von einer Vielzahl von Programmierern über die Jahre ständig erweitert – wobei die Strukturierung oft auf der Strecke blieb. Das Ergebnis sind schwer zu wartende Programme. Von einer Neuprogrammierung wird in der Regel abgesehen, da niemand mehr so genau weiß, was alles integriert ist.

Eine Modernisierung solcher Programme mit Hilfe der ILE-Konzepte ist jedoch möglich.

Ergänzung 29./2014

3.1.4.2 Integrated Language Environment (ILE)

Im Gegensatz zu OPM ermöglicht ILE eine wesentlich modularere Programmierung. Neben Programmen können interne und exportierte Prozeduren und Funktionen erstellt werden.

Interne Prozeduren sind verbesserte Subroutinen, die innerhalb des Programms/Moduls mit Parametern aufgerufen werden können. Innerhalb der internen Prozeduren können Variablen und Dateien definiert werden, die nur in dieser Prozedur verwendet werden können.

Exportierbare Prozeduren können von allen Programmen und anderen Prozeduren aufgerufen werden. Der Source Code wird nur einmal in einer Prozedur hinterlegt, die beliebig aufgerufen werden kann. Ist eine Änderung erforderlich, so muss sie nur an einer einzigen Stelle vorgenommen werden. Damit wird der Wartungsaufwand im Vergleich zu OPM-Programmen beträchtlich reduziert.

Obwohl mit den ILE-Konzepten eine sehr flexible und modulare Programmierung erreicht werden kann, **ist und bleibt RPG prozedural.**

Für die ILE-Programmierung wurde eine Reihe neuer Objekte und Objektarten eingeführt:

Modul:
Kleinste Compile-Einheit, das heißt, aus einer Teildatei wird genau ein Modul erstellt.

Programm:
In ILE-Umgebungen kann sich ein Programm jedoch aus mehreren Modulen zusammensetzen. Programme können direkt aufgerufen werden.

Serviceprogramm:
Verbund von Modulen, die wiederum aus mehreren internen und exportierten Prozeduren bestehen.

Die exportierten Prozeduren können von außen (von anderen Prozeduren/Programmen) aufgerufen werden.

Binder-Verzeichnis:
Verzeichnis, in das Module und Serviceprogramme eingetragen werden können.

Binder-Verzeichnisse können zur Compile-Zeit verwendet werden, um die aufgerufenen Prozeduren in Modulen und Serviceprogrammen zu finden. Die Module und Serviceprogramme werden anschließend in die Programm- oder Serviceprogrammobjekte integriert.

3.1.4.3 Objektorientierte Programmierung (OOP)

Ein Objekt besteht jeweils aus Daten und Funktionen/Methoden. Auf die Daten innerhalb des Objekts kann nur mit Hilfe der integrierten Funktionen/Methoden zugegriffen werden.

Versucht man einen Vergleich mit den ILE-Konstrukten, so kann man sich ein Objekt als globale Datenstruktur in einem Serviceprogramm vorstellen. Die Daten in der globalen Datenstruktur können nur von außen und nur durch den Aufruf der aus dem Serviceprogramm exportierten Prozeduren geändert werden.

Der Unterschied zwischen OOP und ILE liegt darin, dass bei ILE jeweils nur eine einzige „Instanz" des Objekts vorhanden ist. Ein (Service-)Programm kann innerhalb der gleichen Aktivierungsgruppe nur einmal aktiviert werden. Damit ist auch die globale Datenstruktur nur einmal vorhanden.

Bei OOP dagegen kann jederzeit ein neues Objekt generiert, mit Daten versorgt und verwendet werden. Da bei OOP das Objekt immer mit den Daten und den Prozeduren identifiziert wird, können auch Techniken wie Überladung und Vererbung verwendet werden.

Überladung bedeutet, dass der gleiche Prozedur-/Funktions-/Methodenname mit unterschiedlicher Parameterdefinition in der gleichen Klasse mehrfach hinterlegt werden kann. Ebenso kann der gleiche Funktionsname für unterschiedliche Objekte verwendet werden.

In ILE-Umgebungen sollte ein Prozedur-/Funktionsname eindeutig sein. Gleiche Funktions-/Prozedurnamen sind zwar möglich, unter Umständen wird jedoch die falsche Prozedur/Funktion aufgerufen.

Sofern Prozedurnamen mit dem Serviceprogrammnamen qualifiziert werden können, sollte auch in ILE-Umgebungen eine Überladung möglich sein. Vielleicht wird diese Erweiterung noch irgendwann in der Zukunft in die ILE-Konzepte integriert.

Bei Vererbung werden Prozeduren/Funktionen/Methoden von übergeordneten Klassen übernommen und gegebenenfalls modifiziert. Diese Funktionalität wird vermutlich nie Eingang in die ILE-Konzepte finden.

Ergänzung 29./2014

3.1.4.4 Vergleich OPM, ILE und OOP

Im folgenden Schaubild werden die verschiedenen Programmkonzepte anhand eines Beispiels verglichen.

Bei der OPM-Methode auf der linken Seite wird in zwei unabhängigen Subroutinen der Adressstamm eingelesen, einmal für die Liefer- und einmal für die Rechnungsadresse. Der in der ersten Subroutine eingelesene Datensatz wird in der zweiten Subroutine überschrieben. Aus diesem Grund müssen die Liefer-/ Adressdaten gerettet und in Variablen umgeladen werden.

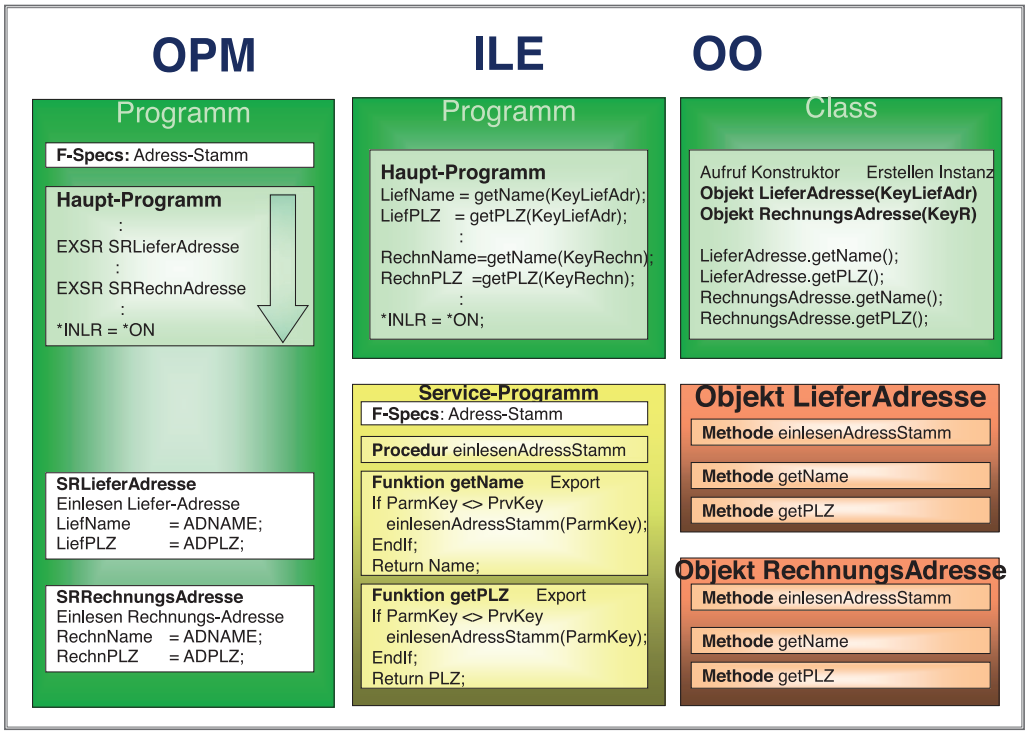
Werden die Liefer- und Rechnungsadresse auch in einem anderen Programm benötigt, so muss der gleiche Quellcode in die andere Programmquelle übernommen werden.

Bei der ILE-Methode in der Mitte wurde der Zugriff auf den Adressstamm in ein eigenes Serviceprogramm ausgelagert. Das Serviceprogramm besteht aus der internen Prozedur „einlesenAdressstamm“, in der der gewünschte Adressstamm in eine globale Datenstruktur eingelesen wird.

Des Weiteren enthält das Serviceprogramm die beiden exportierten Funktionen „getName“ und „getPLZ“. An beide Funktionen wird der Schlüssel für den Zugriff auf den Adressstamm als Parameter übergeben. Innerhalb der Funktionen wird der übergebene Schlüssel mit dem Schlüssel des aktuell eingelesenen Datensatzes verglichen. Sofern der Schlüssel identisch ist, wird nur entweder der Name (bei der Funktion getName) oder die Postleitzahl (bei der Funktion getPLZ) zurückgegeben, ohne dass ein erneuter Dateizugriff erfolgt. Weicht der Schlüssel ab, wird die interne Prozedur „einlesenAddressstamm“ aufgerufen. Die Übergabe des Schlüssels ist erforderlich, da zwischen zwei Funktionsaufrufen für den gleichen Datensatz eine andere Funktion Daten für einen anderen Adressstamm angefordert haben könnte.

Im rufenden Programm ist kein Dateizugriff (mehr) hinterlegt. Die gewünschten Daten werden über Funktionsaufrufe eingelesen. Werden die gleichen Informationen auch in einem anderen Programm benötigt, müssen lediglich die Funktionsaufrufe eingebunden werden.

Bei der OO-Methode wird zur Laufzeit für jede Adresse ein Objekt durch den Aufruf von Constructor-Methoden generiert und der entsprechende Adressstamm eingelesen. Auf die in den Objekten gesicherten Daten kann durch Angabe des Objekts und der gewünschten Methode direkt zugegriffen werden.



Vergleich OPM, ILE, OO

3.1.4
Seite 6

3.1.5 Quellenstruktur

In einer RPG-IV-Teildatei können sowohl der Quellcode für Programme als auch Prozeduren hinterlegt werden.

An dieser Stelle wird zwischen Main- und (Sub-)Procedures unterschieden.

In einer Teildatei können die folgenden Prozedurarten enthalten sein:

- Main-Procedure
- Main-Procedure und (Sub-)Procedures
- Nur interne und exportierte Procedures

3.1.5.1 Main Procedure

Klassische RPG-Programme bestehen lediglich aus einer Main-Procedure oder anders ausgedrückt, die Main-Procedure entspricht der OPM-Programmstruktur.

In der Main-Procedure können die Bestimmungsarten H, F, D, (I), C und (O) codiert werden.

Main-Procedures unterliegen dem RPG-Zyklus, der unter anderm die in den F-Bestimmungen hinterlegten Dateien automatisch öffnet und die Initialisierungs-Subroutine aktiviert. Man spricht in diesem Zusammenhang auch von zyklischen Main-Procedures zur Unterscheidung von linearen Main-Procedures, die mit Release 6.1 eingeführt wurden.

Eine lineare Main-Procedure wird wie eine Prozedur codiert und unterliegt nicht dem RPG-Zyklus, was unter anderem zur Folge hat, dass Programme mit linearer Main-Procedure rekursiv aufgerufen werden können.

Eine Main-Procedure bildet auch den Program-Entry-Point (PEP), das heißt, diese Main-Procedure kann direkt von einer Befehlszeile, einem Menü, einem anderen Programm oder einer anderen Prozedur aus aufgerufen werden.

Das folgende Schaubild zeigt die in einer klassischen Main-Procedure zulässigen Bestimmungsarten.

Innerhalb der globalen D-Bestimmung sollten der Prototyp und das Procedure Interface hinterlegt werden. Prototypen sollten niemals hartcodiert, sondern immer als Copy-Strecke eingebunden werden. *ENTRY PLISTs sollten nicht mehr verwendet werden, da mit Prototyping die Parameter bereits zur Compile-Zeit geprüft und Parameterlisten im RPG-Free-Format nicht unterstützt werden.

Ergänzung 29/2014

3.1.5
Seite 2

Best.-Arten		Bemerkung
H		H-Bestimmungen evt. Angabe Aktivierungsgruppe und Binderverzeichnisse
F		**globale** Datei-Bestimmungen, d.h. gelten für alle (Sub-)Proceduren
I		Nur für Zyklus erforderlich / Umbenennung von Feldern sollte in D-Bestimmungen erfolgen
D	PR	**Prototypen** für **alle verwendeten** oder **definierten Prozeduren** (intern oder exportiert)
		Prototypen für **aufgerufene Programme** (optional / im Free-Format erforderlich)
		Prototypen für exportierte Prozeduren als Copy-Strecken einbinden
	PI	**Procedure-Interface** der Main-Procedure (optional)
		Globale Feld-Definitionen / Verwendung in der Main- und in allen Sub-Procedures
C		C-Bestimmungen der **Main-Procedure**
		Subroutinen können nur innerhalb der **Main-Prozedure** aufgerufen werden
O		Intern beschriebene **Printerfiles** nur **globale** Verwendung
P FirstProc	B	**Beginn der ersten Prozedur**
		Schlüssel-Wort **EXPORT**: Aufruf kann von außerhalb des Programms erfolgen
F		**Lokale** Datei-Bestimmungen, d.h. gelten nur innerhalb der Prozedur
D	PI	**Procedure Interface**
		Lokale Feld-Definitionen / Verwendung nur innerhalb der Sub-Procedure
		Datenstrukturen für lokalen Datei-Zugriff
C		C-Bestimmungen der Sub-Procedure
		Subroutinen sind zulässig und können nur innerhalb der Sub-Procedure verwendet werden
P FirstProc	E	**Ende der ersten Prozedur**
P NextProc	B	Beginn der nächsten Prozedur
D	PI	Procedure Interface
		Lokale Feld-Definitionen / Verwendung nur innerhalb der Prozedur
C		C-Bestimmungen der Prozedur
P NextProc	E	Ende der nächsten Prozedur
**		**Compile Time Data**

Aufbau Main-Procedure

Das folgende Beispiel zeigt das Gerüst für eine Main-Procedure.

Im Quellcode, genauer gesagt in den H-Bestimmungen, ist hinterlegt, dass das Programm nicht in der Default-Aktivierungsgruppe, sondern in der Aktivierungsgruppe MYPGM erstellt werden soll. Diese Angaben überschreiben die entsprechenden Optionen im Compile-Befehl CRTBNDRPG. Die Angaben sind durch Compiler-Direktiven bedingt und werden nur verwendet, wenn die Kompilierung über den CL-Befehl CRTBNDRPG erfolgt. Erfolgt eine zweistufige Kompilierung, das heißt, wird das Modul mit dem CL-Befehl CRTRPGMOD erstellt und anschließend über den CL-Befehl CRTPGM eingebunden, so werden die Angaben ignoriert.

Ebenso ist in den H-Bestimmungen eine Reihe von Binder-Verzeichnissen hinterlegt, in denen sich die zu bindenden Module und Serviceprogramme befinden. Binder-Verzeichnisse werden zur Compile-Zeit durchforstet, um die aufgerufenen Prozeduren zu lokalisieren und die entsprechenden Module oder Signaturen der Serviceprogramme in die (Service-)Programmobjekte einzubinden.

Die Prototypen der aufgerufenen Prozeduren werden über Copy-Strecken in die Quelle integriert.

Das Procedure Interface (Ersatz für die *ENTRY PLIST) wird in den globalen D-Bestimmungen hinterlegt.

Anschließend werden die globalen D-Bestimmungen definiert und die C-Bestimmungen wie gewohnt programmiert.

Da Programme mit zyklischen Main-Procedures dem Zyklus unterliegen, muss die Main-Procedure entweder mit *INLR oder RETURN beendet werden.

```
 /If Defined (*CRTBNDRPG)
H DFTACTGRP(*NO) ACTGRP('MYPGM')
 /EndIf
H BNDDIR('MYBNDDIR1': 'MYBNDDIR2': 'QC2LE')
 *----------------------------------------------------------------
 * Global File Definitions - F-Specs
 *----------------------------------------------------------------

 *----------------------------------------------------------------
 * Copy Members - Containing Reference Fields / Constants / Prototypes
 *----------------------------------------------------------------
 /INCLUDE QPROLESRC,MyProto1
 /INCLUDE QPROLESRC,MyProto2

D MyPGM           PI
D
 *----------------------------------------------------------------
 * Global variables
 *----------------------------------------------------------------


 ****************************************************************
 * M A I N   P R O C E D U R E  - Global C-Specs
 ****************************************************************
 /FREE


  *INLR = *ON;
  Return;
  //--------------------------------------------------------------
  // Global Subroutines
  //--------------------------------------------------------------
 /END-FREE
```

Gerüst einer Main-Procedure

3.1.5.2 (Sub-)Procedures

Im Anschluss an die Main-Procedure können beliebig viele Prozeduren codiert werden.

In Quellen, in denen Prozeduren codiert werden, ist eine zyklische Main-Procedure nicht zwingend erforderlich. Die erste Prozedur wird unmittelbar nach den globalen D-Bestimmungen codiert. Um zu verhindern, dass der Compiler versucht den RPG-Zyklus zu integrieren, muss in den H-Bestimmungen entweder das Schlüsselwort NoMain (sofern die Quelle nur interne und exportierte Prozeduren enthält) oder das Schlüsselwort Main (wenn eine lineare Main-Procedure verwendet wird) angegeben werden.

Im Gegensatz zu der Main-Procedure unterliegen (Sub-)Procedures nicht dem RPG-Zyklus und können somit rekursiv aufgerufen werden.

Prozeduren beginnen und enden immer mit einem P-Statement. Werden die neuen Free-Format-Operations-Codes verwendet, so beginnt eine Procedure mit dem Operations-Code DCL-PROC und endet mit dem Operations-Code END-PROC. Innerhalb der Prozedur können die Bestimmungsarten F, D und C codiert werden.

Das folgende Beispiel zeigt das Format der P-Bestimmungen sowie den Beginn der internen Prozedur IntFirstProc.

Unmittelbar nach der Bestimmungsart P kann der Prozedurname zwischen Position 7 und Position 21 angegeben werden. Bei längeren Prozedurnamen kann über Position 21 fortgeschrieben und können die übrigen Angaben in der nächsten Zeile angegeben werden. In diesem Fall muss der Umbruch jedoch durch drei aufeinanderfolgende Punkte (…) gekennzeichnet werden.

Bei dem beginnenden P-Statement ist die Angabe des Prozedurnamens zwingend erforderlich, bei dem beendenden P-Statement ist die Angabe des Prozedurnamens optional.

Auf Position 25 wird angegeben, ob es sich um ein beginnendes (B=Beginn) oder beendendes (E=End) P-Statement handelt.

Ab Position 44 können Schlüsselworte angegeben werden.

```
.DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++++
 D Cvt2Lower        PI                50A   Varying
 D   ParText                          50A   Varying Options(*Trim) Const
```

P-Bestimmungen

Mit Release 7.1 und Technology Refresh 7 wurden für die fixen Bestimmungsarten H, D, F und P auch Free-Format-Alternativen bereitgestellt. Im Free-Format beginnt eine Prozedur mit dem Operations-Code DCL-PROC und endet mit dem Operations-Code END-PROC. Der Prozedurname wird jeweils unmittelbar nach dem Operations-Code angegeben, wobei die Angabe bei dem Ende-Operations-Code optional ist. Im Anschluss an den Prozedurnamen können die gewünschten Schlüsselworte angegeben werden.

Das folgende Beispiel zeigt den Beginn und das Ende der internen Prozedur IntSecondProc im Free-Format.

```
DCL-PROC IntSecondProc;

END-Proc;
```

Prozedurdefinition im Free-Format

Die folgende Abbildung zeigt den Aufbau und die zulässigen Bestimmungsarten für zwei (Sub-)Procedures.

Best.-Arten		Bemerkung
P FirstProc	B	**Beginn der ersten Prozedur**
		Schlüssel-Wort **EXPORT**: Aufruf kann von außerhalb des Programms erfolgen
F		**Lokale** Datei-Bestimmungen, d.h. gelten nur innerhalb der Prozedur
D	PI	**Procedure Interface**
		Lokale Feld-Definitionen / Verwendung nur innerhalb der Sub-Procedure
		Datenstrukturen für lokalen Datei-Zugriff
C		C-Bestimmungen der Sub-Procedure
		Subroutinen sind zulässig und können nur innerhalb der Sub-Procedure verwendet werden
P FirstProc	E	**Ende der ersten Prozedur**
P NextProc	B	Beginn der nächsten Prozedur
D	PI	Procedure Interface
		Lokale Feld-Definitionen / Verwendung nur innerhalb der Prozedur
C		C-Bestimmungen der Prozedur
P NextProc	E	Ende der nächsten Prozedur
**		**Compile Time Data**

Aufbau/Bestimmungsarten von (Sub-)Procedures

Der Aufruf von Sub-Procedures erfolgt über den Operations-Code CALLP (= Call with Prototype). Wird im Free-Format codiert, so ist die Angabe des Operations-Codes CALLP nur erforderlich, wenn eine Erweiterung angegeben werden soll – zum Beispiel (E), um einen Fehler abzufangen.

Für Prozeduren kann auch ein Rückgabewert definiert werden, in diesem Fall spricht man von Funktionen. Funktionen werden genau wie reguläre Built-In-Funktionen aufgerufen, zum Beispiel in einem EVAL-Statement (im Free-Format optional), in einem Ausdruck, in einer IF-Anweisung oder als Parameterfeld bei einem Programm-/Prozeduraufruf etc.

3.1.5
Seite 6

(Sub-)Procedures können mit Ein-/Ausgabeparameter definiert und aufgerufen werden. Die Parameterdefinition muss über Prototyping erfolgen, das heißt, das Procedure-Interface muss in den lokalen D-Bestimmungen der (Sub-)Procedure hinterlegt werden. Für interne Prozeduren ist die Angabe des Prototypen seit Release 7.1 optional. Für exportierte Prozeduren ist die Angabe des Prototypen zwingend erforderlich. Prototypen sollten immer in einer eigenen Teildatei hinterlegt werden, die in Form einer Copy-Strecke integriert wird.

Dateien, die in den lokalen F-Bestimmungen definiert wurden, und Variablen, die in den prozedurinternen D-Bestimmungen (lokal) definiert wurden, können nur innerhalb der Prozedur verwendet werden. Aus einer Prozedur kann jedoch auf alle Dateien und Variablen, die in den globalen F- beziehungsweise D-Bestimmungen definiert wurden, zugegriffen werden.

Wird die gleiche Variable einmal global und einmal lokal definiert, so kann innerhalb der Prozedur nur auf die lokale Variable zugegriffen werden. In anderen Prozeduren in der gleichen Teildatei, in denen die Variable nicht lokal definiert wurde, wird auf die globale Variable zugegriffen.

Da (Sub-)Procedures nicht dem RPG-Zyklus unterliegen, wird jedem Aufruf ein neues Variablenset zur Verfügung gestellt, das heißt, die lokalen Variablen sind/werden beim Aufruf immer initialisiert. Sofern jedoch der Inhalt von lokalen Variablen über mehrere Aufrufe hinweg gerettet werden soll, kann bei der Definition der entsprechenden Variable das Schlüsselwort STATIC angegeben werden.

Da (Sub-)Procedures nicht dem RPG-Zyklus unterliegen, werden für lokale F-Bestimmungen keine I/O-Strukturen generiert, was zur Folge hat, dass für die zu verarbeitenden Datensätze Datenstrukturen angelegt werden müssen. Diese Datenstrukturen können entweder als externe Datenstrukturen mit Hilfe des Schlüsselworts EXTNAME oder basierend auf dem Dateiformat mit Hilfe des Schlüsselworts LIKEREC definiert werden. Diese Datenstrukturen müssen dann in Verbindung mit den Schreib-/Lese-Operations-Codes – zum Beispiel READ, CHAIN, UPDATE – angegeben werden, um den Datensatz aufzunehmen.

Die Codierung innerhalb der lokalen C-Bestimmungen unterscheidet sich nicht von der Codierung in Main-Procedures, das heißt, es können alle zulässigen Operations-Codes und Built-in-Funktionen, aber auch embedded SQL verwendet werden.

Aus (Sub-)Procedures können nur andere (Sub-)Procedures oder Programme aufgerufen werden. Es ist jedoch nicht möglich aus einer (internen) Sub-Procedure eine Subroutine in der Main-Procedure aufzurufen.

Innerhalb von (Sub-)Procedures können – wie in der Main-Procedure auch – Subroutinen codiert werden. Subroutinen werden am Ende der prozedurinternen C-Bestimmungen vor dem beendenden P-Statement angegeben. Dies ist insbesondere am Anfang hilfreich, da auf diese Art und Weise auch etwas größere Prozeduren strukturiert werden können. Sobald man jedoch gelernt hat, richtig modular zu denken, wird man feststellen, dass Subroutinen tatsächlich so gut wie nicht mehr benötigt werden.

(Sub-)Procedures werden mit dem Operations-Code RETURN verlassen. Da der RPG-Zyklus jedoch nicht eingebunden wird, wird nicht geprüft, ob er in der (Sub-)Procedure auch wirklich angegeben wurde. Fehlt der OpCode RETURN, so wird die Prozedur nach Ausführung des letzten Statements beendet. Dennoch sollte man es sich zur Gewohnheit machen, den Operations-Code RETURN am Ende der Prozedur anzugeben. Dies ist insbesondere dann wichtig, wenn in der Prozedur noch Subroutinen codiert sind. Fehlt der RETURN-Operations-Code in diesem Szenario, wird das Prozedurende nie erreicht.

Ergänzung 29./2014

3.1.5.2.1 Interne Prozeduren

Interne Prozeduren können nur innerhalb des Moduls beziehungsweise innerhalb der gleichen Teildatei verwendet werden.

Das folgende Beispiel zeigt den Source Code für ein Programm, in dem die interne Prozedur „CvtUmlauteProc“ definiert und aufgerufen wird. In der internen Prozedur „CvtUmlauteProc“ werden die Umlaute ä, ö, ü, ß in ae, oe, ue und ss konvertiert.

Da es sich um eine interne Prozedur handelt, ist die Angabe des Prototypen in den globalen D-Bestimmungen optional und fehlt daher auch in dem Beispiel. Deshalb werden in den globalen D-Bestimmungen nur die beiden Varibalen „Text“ und „CvtText“ definiert.

In der Main-Procedure wird die Variable „Text“ zunächst initialisiert. Im Anschluss wird die interne Prozedur „CvtUmlauteProc“ aufgerufen (1). Da der Aufruf im Free-Format erfolgt, ist die Angabe des Operations-Codes CALLP an dieser Stelle nicht zwingend erforderlich. Das Ergebnis beziehungsweise der Wert des Ausgabeparameters wird mit Hilfe des Operations- Codes DSPLY angezeigt.

Die Prozedur „CvtUmlauteProc“ erwartet zwei Parameter. Beim ersten Parameter handelt es sich um einen Ausgabeparameter, in dem der konvertierte Text zurückgegeben wird (2a). Beim zweiten Parameter handelt es sich um einen Input-Only-Parameter, was durch die Angabe des Schlüsselworts CONST festgelegt wird. Die Angabe des Schlüsselworts OPTIONS(*TRIM) bewirkt, dass führende und folgende *BLANKS von dem eingehenden Text entfernt werden (2b).

In der Datenstruktur DSCVT beziehungsweise in den untergeordneten Feldgruppen sind die Umlaute sowie deren Übersetzung hinterlegt (3). Das Schlüsselwort STATIC bewirkt, dass der Inhalt der Datenstruktur auch nach Beendigung der Prozedur erhalten bleibt.

In den lokalen C-Bestimmungen wird zunächst geprüft, ob überhaupt ein Text übergeben wurde. Sofern kein Text übergeben wurde, wird die Prozedur sofort verlassen.

In einer Schleife werden die Umlaute aus den Feldgruppen verarbeitet, die in der Datenstruktur DSCVT hinterlegt sind, und mit Hilfe der Built-In-Function %SCANRPL im Ausgabeparameter ersetzt. Nach der Umsetzung wird die Prozedur mit RETURN verlassen.

```
D Text            S             50A   Varying
D CvtText         S             50A   Varying
 *---------------------------------------------------------
 /Free

    Text = 'Der Bär frisst Käse und löffelt Blütenhonig';
    CvtUmlauteProc(CvtText: Text);
(1)
    Dsply CvtText;

   *INLR = *On;
 /End-Free
 //*********************************************************
 // Procedure Name:  CvtUmlauteProc
 // Purpose:         Umlaute (ä,ü,ö,ß into ae, oe, ue, ss)
    konvertieren
 // Return Value:
 // Parameter:       POutText => konvertierter Text
 // Parameter:       ParText  => zu konvertierender Text
 //*********************************************************
P CvtUmlauteProc...
P                 B

D CvtUmlauteProc...
D                 PI
D   POutText                50A   Varying
(2a)
D   ParText                 50A   Varying Options(*Trim) Const
(2b)

D DSCvt           DS              Qualified         Static (3)
D                           21A   inz('äaeöoeüueßssÄAeÖOeÜUe')
D   ArrAll                              Overlay(DSCvt) Dim(7)
D     ArrUml                 1A   Overlay(ArrAll)
D     ArrCvt                 2A   Overlay(ArrAll: *Next)

D Idx             S          3U 0
```

Interne Prozedur – CvtUmlauteProc

3.1.5
Seite 10

```
 *-------------------------------------------------------------
 /Free
   POutText = ParText;
   If %Len(ParText) = *Zeros;
      Return;
   EndIf;

   For Idx = 1 to %Elem(DSCvt.ArrAll);
     POutText = %ScanRpl(DSCvt.ArrUml(Idx): DSCvt.
ArrCvt(Idx): POutText);
   EndFor;

   Return;
 /End-Free
P CvtUmlauteProc...
P                 E
```

Exportierte Prozeduren

Exportierte Prozeduren können auch von außerhalb der Teildatei beziehungsweise von außerhalb des Moduls aufgerufen werden.

Damit eine Prozedur exportiert werden kann, ist zunächst die Angabe des Schlüsselworts EXPORT in der beginnenden P-Bestimmung erforderlich. Wird anstatt der P-Bestimmungen der Free-Format Operations-Code DCL-PROC verwendet, so muss das Schlüsselwort EXPORT bei diesem Operations-Code angegeben werden.

Exportierte Prozeduren sollten nach Funktionalität in Teildateien zusammengefasst werden. In diesen Teildateien sollte weder eine zyklische noch eine lineare Main-Procedure enthalten sein. Um dem Compiler mitzuteilen, dass der RPG-Zyklus nicht integriert werden soll, ist die Angabe des Schlüsselworts NOMAIN in den H-Bestimmungen erforderlich.

Die Teildatei wird anschließend über CRTMODRPG (RPG-Modul erstellen) kompiliert und mit dem Befehl CRTSRVPGM (Serviceprogramm erstellen) oder dem Befehl UPDSRVPGM (Serviceprogramm aktualisieren) in ein Serviceprogramm gebunden. Beim Erstellen oder Ändern des Serviceprogramms kann festgelegt werden, dass alle Prozeduren exportiert werden. Anstatt alle Prozeduren zu exportieren, kann mit Hilfe der Binder-Sprache genau festgelegt werden, welche Prozeduren exportiert werden können und dürfen. Die Binder-Sprache wird in einer Teildatei mit der Attributart BND hinterlegt. Diese Teildatei wird im CRTSRVPGM-Befehl angegeben.

Das folgende Code-Beispiel zeigt die gleiche Konvertierungsprozedur „CvtUmlauteProc“ wie im vorhergehenden Beispiel. Im Gegensatz zum vorherigen Beispiel handelt es sich in diesem Fall jedoch nicht um eine interne, sondern um eine exportierte Prozedur.

Die Änderungen im Source Code sind minimal. Da die Prozedur in einer separaten Teildatei hinterlegt wurde, in der sich nur interne und exportierte Prozeduren befinden, muss in den H-Bestimmungen das Schlüsselwort NoMain (1) angegeben werden.

Die Parameter für Prozeduren müssen mit Prototyping definiert werden. Innerhalb der Prozedur muss das Procedure Interface (vergleichbar mit einer *ENTRY PLIST) hinterlegt werden. Bei exportierten Prozeduren muss der Prototyp in den globalen D-Bestimmungen hinterlegt werden. In diesem Beispiel befindet sich der Prototyp in der Teildatei STRING der Quellendatei QPROLESRC. Diese Teildatei wurde als Copy-Strecke mit Hilfe der Compiler-Anweisung /INCLUDE eingebunden (2).

Copy-Strecken können wahlweise über /INCLUDE oder /COPY eingebunden werden. Der Unterschied besteht lediglich darin, dass Copy-Strecken, die mit /COPY integriert wurden, vom SQL-Precompiler aufgelöst werden, während der SQL-Precompiler Copy-Strecken, die mit /INCLUDE hinterlegt wurden, ignoriert.

Innerhalb der Prozedur gibt es zum vorhergehenden Beispiel nur eine kleine Änderung: Das Schlüsselwort EXPORT wird in der P-Bestimmung angegeben, die den Anfang der Prozedur kennzeichnet (3).

```
 H NoMain
(1)
  *-------------------------------------------------------
  * Prototypes
  *-------------------------------------------------------
  /INclude QPROLESRC,String
(2)
  *-------------------------------------------------------
  * Global Definitions
  *-------------------------------------------------------

   //*****************************************************
   // Procedure Name:  CvtUmlauteProcE
   // Purpose:         Convert German Umlaute
                       (ä,ü,ö,ß into ae, oe, ue, ss)
   // Return Value:
```

3.1.5
Seite 12

```
   // Parameter:       ParText  => String to be converted
   // Parameter:       POutText => Revised String
   //*************************************************************
 P CvtUmlauteProcE...
 P                 B                   Export
(3)

 D CvtUmlauteProcE...
 D                 PI
 D   POutText                    50A   Varying
 D   ParText                     50A   Varying Options(*Trim)
Const

 D DSCvt           DS                  Qualified
Static
 D                               21A   inz('äaeöoeüueßssÄAeÖO
eÜUe')
 D   ArrAll                            Overlay(DSCvt) Dim(7)
 D     ArrUml                     1A   Overlay(ArrAll)
 D     ArrCvt                     2A   Overlay(ArrAll: *Next)

 D Idx             S              3U 0
  *-------------------------------------------------------------
  /Free
    POutText = ParText;
    If %Len(ParText) = *Zeros;
       Return;
    EndIf;

    For Idx = 1 to %Elem(DSCvt.ArrAll);
      POutText = %ScanRpl(DSCvt.ArrUml(Idx): DSCvt.
ArrCvt(Idx): POutText);
    EndFor;

    Return;
  /End-Free
 P CvtUmlauteProcE...
 P                 E
```

Exportierte Prozedur – CvtUmlatuProcE

Nachdem die Prozedur in ein Serviceprogramm gebunden wurde, kann sie von überall aus aufgerufen werden.

Das folgende Beispiel zeigt ein Programm, in dem die exportierte Prozedur „CvtUmlautProcE“ (5) aufgerufen wird.

In den H-Bestimmungen wird festgelegt, dass das Programm nicht in der Default-Aktivierungsgruppe, sondern in der Aktivierungsgruppe MYPGM ausgeführt wird (1). Diese Schlüsselworte sind durch Compiler-Direktiven bedingt und werden nur berücksichtigt, wenn das Programm mit dem Compile-Befehl CRTBNDRPG erstellt wird (2).

Das Serviceprogramm STRING, in dem sich die Prozedur befindet, wurde im Binder-Verzeichnis MYBNDDIR hinterlegt (3). Durch diese Einträge in den H-Bestimmungen ist eine zweistufige Kompilierung des Programms (Erstellen des Moduls und anschließendes Binden in ein Programm) nicht zwingend erforderlich. Stattdessen kann das Programm mit dem Befehl CRTBNDRPG umgewandelt werden.

Da in dem Beispiel eine externe Prozedur aufgerufen wird, muss der Prototyp der Prozedur hinterlegt werden. Der Prototyp wurde in der Teildatei STRING der Quellendatei QPROLESRC hinterlegt und über die Compiler-Anweisung /INCLUDE integriert (4).

Die Prozedur kann jetzt direkt ausgeführt werden.

Ergänzung 29./2014

3.1.5
Seite 14

```
  /If Defined (*CrtBndRPG)
(2)
 H  DftActGrp(*No) ActGrp('MYPGM')
(1)
  /EndIf
 H Bnddir('MYBNDDIR')
(3)
  *---------------------------------------------------------------
  * Prototypes
  *---------------------------------------------------------------
  /INclude QPROLESRC,String
(4)
  *---------------------------------------------------------------
  * Global Variables
  *---------------------------------------------------------------
 D Text            S             50A   Varying
 D CvtText         S             50A   Varying
  *---------------------------------------------------------------
  /Free
     Text = 'Der Bär frißt Käse und löffelt Blütenhonig';
     CvtUmlauteProcE(CvtText: Text);
(5)
     Dsply CvtText;

     *INLR = *On;
     Return;
  /End-Free
```

Aufruf einer externen Prozedur

3.1.5.2.3 Funktionen

Funktionen sind interne oder exportierte Prozeduren mit Rückgabewert.

Der Aufruf von Funktionen erfolgt nicht über den Operations-Code CALLP. Stattdessen werden Funktionen wie Built-In-Funktionen behandelt, das heißt, sie können in einem EVAL-Statement, in einer IF-Anweisung, in einem Ausdruck oder als Parameter bei einem Programm oder Prozeduraufruf angegeben werden.

Der Rückgabewert wird im Prototyp und Procedure Interface in der PR- beziehungsweise PI-Zeile durch die Angabe eines Datentyps und gegebenenfalls einer Länge definiert. Wird der Prototyp in der Free-Format-Schreibweise definiert, so wird der Rückgabewert nach dem Prozedurnamen in den Operations-Codes DCL-PR (Declare Prototype) und DCL-PI (Declare Procedure Interface) in Form eines Datentyps und gegebenenfalls einer Länge angegeben.

Der Rückgabewert wird im Faktor 2 des Operations-Codes RETURN hinterlegt und so an die rufende Prozedur zurückgegeben. Bei Funktionen ist die Angabe des Operations-Code RETURN zwingend erforderlich.

Im folgenden Beispiel werden die internen Funktionen „cvt2Lower“ (Text in Kleinbuchstaben konvertieren) (1) und „cvt2Upper“ (Text in Großbuchstaben konvertieren) (2) definiert. Beide Funktionen werden aus der Main-Procedure (3) heraus aufgerufen.

In dem Code-Beispiel wurde die eine Funktion mit fixen P- und D-Bestimmungen codiert, während die zweite Funktion komplett im Free-Format codiert wurde. Seit Release 7.1 und Technology Refresh 7 müssen die lästigen /FREE- und /END-FREE-Direktiven nicht mehr angegeben werden, und es kann zwischen fixem und freiem Format beliebig gewechselt werden.

Durch die Free-Format H-Bestimmung wird festgelegt, dass das Programm in der Aktivierungsgruppe STRING ausgeführt wird. Die Angabe der H-Bestimmung DFTACTGRP ist nicht länger erforderlich, da sie seit Release 7.1 TR 7 automatisch gesetzt wird, wenn eines der Schlüsselworte ACTGRP, BNDDIR, STGMDL in den H-Bestimmungen hinterlegt wurde.

In den globalen D-Bestimmungen werden die Variablen „Text“, „UpperText“ und „LowerText“ jeweils alphanumerisch mit einer variablen Länge von maximal 50 Zeichen definiert.

In der Main-Procedure wird zunächst die Variable „Text“ initialisiert. Anschließend wird zunächst die Funktion „Cvt2Upper“ aufgerufen, der Rückgabewert in die Variable „UpperText“ ausgegeben und anschließend mit dem Operations-Code DSPLY angezeigt. Im Anschluss wird der gleiche Text mit der Funktion „Cvt2Lower“ in Kleinschrift konvertiert, in der Variable „LowerText“ ausgegeben und ebenfalls mit DSPLY angezeigt.

Ergänzung 29./2014

3.1.5
Seite 16

Die Funktion „Cvt2Lower“ wurde in der klassischen Fix-Format-Schreibweise codiert, während für die Funktion „Cvt2Upper“ die Free-Format-Schreibweise verwendet wurde.

Im Procedure-Interface, das jeweils nach dem beginnenden P-Statement beziehungsweise nach dem Operations-Code DCL-PROC angegeben wurde, wird der Rückgabewert durch den Datentyp A Varying beziehungsweise VARCHAR mit einer Maximallänge von 50 Zeichen definiert.

Innerhalb der Prozeduren sind jeweils lokale Konstanten definiert, mit deren Hilfe die Konvertierung in Groß- beziehungsweise Kleinschrift erfolgt.

Die Konvertierung erfolgt jeweils mit Hilfe der Built-In-Funktion %XLATE, die im Faktor 2 des Operations-Codes Return angegeben wurde (4).

```
  /If Defined (*CrtBndRPG)
    CTL-OPT ActGrp('STRING');
  /EndIf
  *-------------------------------------------------------------
  * Global Definitions
  *-------------------------------------------------------------
 D Text            S             50A   Varying

   DCL-S UpperText VarChar(50);
   DCL-S LowerText VarChar(50);
  *-------------------------------------------------------------

     Text = 'Der Bär frisst Käse und löffelt Blütenhonig';

     UpperText = Cvt2Upper(Text);
(3)
     Dsply UpperText;

     LowerText = Cvt2Lower(Text);
(3)
     Dsply LowerText;

    *INLR = *On;
  ************************************************************
  * Procedure Name:  Cvt2Lower
  * Purpose:         Text in Kleinschrift konvertieren
  * Return Value:    Konvertierter Text
  * Parameter:       ParText => zu konvertierender Text
  ************************************************************
```

```
 P Cvt2Lower       B
(1)

 D Cvt2Lower       PI            50A   Varying
 D   ParText                     50A   Varying Options(*Trim)
Const

 D constUpper      C
Const('ABCDEFGHIJKLMNOPQRSTU+
 D                                       VWXYZÄÖÜ')
 D constLower      C
Const('abcdefghijklmnopqrstu+
 D                                       vwxyzäöü')
  *-------------------------------------------------------
  /Free
    Return %XLate(ConstUpper: ConstLower: ParText);
(4)
  /End-Free
 P Cvt2Lower       E
  ********************************************************
  * Procedure Name:  Cvt2Upper
  * Purpose:         Text in Großschrift konvertieren
  * Return Value:    Konvertierter Text
  * Parameter:       ParText => zu konvertierender Text
  ********************************************************
   DCL-Proc Cvt2Upper;
(2)

     DCL-PI *N VarChar(50);
            ParText VarChar(50);
     End-Pi;

     DCL-C ConstUPPER Const('ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ');
     DCL-C ConstLower Const('abcdefghijklmnopqrstuvwxyzäöü');
   //-------------------------------------------------------
     Return %Xlate(ConstLower: ConstUpper: ParText);
(4)
   End-Proc;
```

Definition und Aufruf von Funktionen

3.1.5.2.4 Globale und lokale Definitionen

Dateien und Variablen können sowohl global als auch lokal definiert werden. Die globale Definition erfolgt entweder vor den C-Bestimmungen der Main-Procedure oder, sofern keine zyklische Main-Procedur vorhanden ist, vor der Codierung der ersten Prozedur.

Dateien und Variablen, die in den globalen F- oder D-Bestimmungen definiert sind, können sowohl in der Main-Procedure als auch in den internen und exportierten Prozeduren verwendet und modifiziert werden. Solange man nur mit einem Programm und nur mit internen Prozeduren arbeitet, sind globale Definitionen und deren Verwendung weniger kritisch. Wenn jedoch exportierte Prozeduren auf globale Variablen zugreifen, wird das Ganze schon schwieriger. Da die Prozeduren beliebig von außerhalb aufgerufen werden können, ist die Reihenfolge in der der Aufruf erfolgen kann nicht mehr zwingend vorgegeben. Damit können Globale Variablen, durch eine falsche Aufruffolge fälschlich gesetzt oder überschrieben werden.

Beispiel: Eine Prozedur könnte einen globalen Wert setzen, der zu einem späteren Zeitpunkt von einer anderen Prozedur weiterverwendet werden soll. Vor dem Aufruf der zweiten Prozedur wird jedoch eine dritte Prozedur aus einem Unterprogramm aufgerufen, die den zuvor gesetzten globalen Wert überschreibt. Wird jetzt die zweite Prozedur aufgerufen, enthält die Variable nicht mehr den von der ersten Prozedur gesetzten Wert, sondern den durch die dritte Prozedur geänderten Wert, was zu unerwarteten Ergebnissen führen kann.

Fazit:

Bei der Arbeit mit Prozeduren, unabhängig davon ob diese nur intern verwendet werden oder auch von außen aufgerufen werden können, sollte man nur in Ausnahmefällen mit globalen Variablen und Dateien arbeiten.

Lokale Definitionen und die Übergabe über Parameter sind die bessere Option.

Je besser die Prozeduren gekapselt sind, desto einfacher sind sie zu handeln.

Lokale Definitionen erfolgen innerhalb der Prozeduren in den lokalen F- und D-Bestimmungen. Dateien und Variablen, die lokal definiert sind, können nur innerhalb der Prozedur verwendet werden. Dies gilt auch dann, wenn eine Variable mit dem gleichen Namen einmal global und einmal lokal definiert wurde. Innerhalb der Prozedur kann nur mit der lokalen Variablen gearbeitet werden. Änderungen der lokalen Variable haben keinen Einfluss auf die globale Variable.

Beispiel: Die Variable „Index“ ist sowohl global als auch lokal definiert.

In der Main-Procedure wird eine Feldgruppe über eine Schleife verarbeitet. Als Zähler wird die Variable „Index“ verwendet. Für jedes Element wird die Prozedur XY aufgerufen.

In dieser Prozedur wird ebenfalls in einer Schleife eine weitere Feldgruppe verarbeitet. Auch hier wird die Variable „Index“, die auch lokal definiert ist, als Zähler verwendet.

Wird die Prozedur XY für den globalen Indexwert 2 aufgerufen, und nimmt bei Beendigung der Prozedur XY die lokale Indexvariable den Wert 17 ein, so wird beim nächsten Durchlauf der globale Indexwert dennoch auf 3 gesetzt.

Tipp:

Die Namenskonventionen sollten so festgelegt werden, dass globale und lokale Variablen unterschieden werden können. Zum Beispiel beginnen alle lokalen Variablen mit l oder loc oder l_, während die globalen Variablen entweder kein Präfix haben oder mit g oder gbl oder g_ beginnen.

Ergänzung 29./2014

3.1.5
Seite 20

3.1.6 Prototyping-Basics

Prototyping wurde als neue Form der Parameterübergabe speziell für Prozeduren eingeführt. Neben der reinen Definition ermöglicht Prototyping eine Parameterprüfung zur Compile-Zeit. Daneben erlaubt Prototyping die Definition von reinen Input-Only- und optionalen Parametern.

Erst mit Release V5R1 hat man Prototyping auch für Programme zugelassen und kann somit die Vorteile, die Prototyping gegenüber den klassischen *ENTRY PLISTs und anderen Parameterlisten hat, auch für Programme nutzen.

Um eine Parameterprüfung zur Compile-Zeit zu ermöglichen, werden sowohl ein Procedure Interface (Eingangsparameter) als auch ein Prototyp (zum Abgleich der Parameter) benötigt.

Procedure Interface:
Das Procedure-Interface entspricht der Eingangsparameterliste, wird jedoch in den D- und nicht in den C-Bestimmungen hinterlegt.

Prototyp:
Beim Prototyp handelt es sich um ein Duplikat des Procedure-Interface mit einigen wenigen Abweichungen.

Der Prototyp muss in der Quelle eingebunden werden, in der die Prozedur oder das Programm codiert ist, damit der Compiler einen ersten Abgleich machen kann.

Des Weiteren muss der Prototyp in alle Quellen eingebunden werden, in denen die Prozedur aufgerufen wird. Dadurch kann der Compiler auch beim Umwandeln der rufenden Prozeduren die Parameter abgleichen.

Anmerkung:

Da die Parameter nur mit den Prototypen, nicht jedoch mit der echten Prozedur oder dem Programm abgeglichen werden, muss sichergestellt werden, dass der Prototyp auch der tatsächlichen Parameterdefinition entspricht.

Aus diesem Grund sollten Prototypen niemals hart in den Quellen hinterlegt werden, sondern immer in einer eigenen Teildatei gespeichert und als Copy-Strecke mit Hilfe der /COPY- oder /INCLUDE-Direktive in die Quellen eingebunden werden.

3.1.6
Seite 2

3.1.6.1 Procedure Interface – Eingangsparameter

Im Procedure Interface, das bei Main-Procedures in den globalen und bei (Sub-)Prozeduren in den lokalen D-Besimmungen hinterlegt ist, werden die Eingangsparameter definiert.

In der klassischen Schreibweise wird im Procedure-Interface der Prozedurname auf den Positionen 7 – 21 hinterlegt. Bei längeren Namen kann über Position 21 hinausgeschrieben werden und können die übrigen Angaben in der Folgezeile hinterlegt werden. Der Umbruch muss mit drei aufeinanderfolgenden Punkten (…) gekennzeichnet werden.

Auf den Positionen 24–25 muss fix PI (= Procedure Interface) angegeben werden.

Sofern die Prozedur einen Rückgabewert hat, wird dieser in der gleichen Zeile definiert. Dazu werden ein Datentyp, die Länge (sofern für den Datentypen erforderlich) und eventuell zusätzliche Schlüsselworte (zum Beispiel Varying oder DatFmt) angegeben. Rückgabewerte können auch mit Hilfe der Schlüsselworte LIKE (wie eine andere Variable), LIKEDS (wie eine Referenzdatenstruktur) und LIKEREC (wie ein Satzformat) definiert werden.

In den Folgezeilen werden die einzelnen Parameter definiert. Der Parametername wird auf den Positionen 7–21 hinterlegt. Die Positionen 24–25 bleiben leer. Der Parameter wird durch die Angabe des Datentypen, der Länge (sofern der Datentyp dies erfordert) sowie durch zusätzliche Schlüsselworte definiert. Alternativ kann der Parameter über eines der Schlüsselworte LIKE, LIKEDS oder LIKEREC definiert werden.

Das folgende Beispiel zeigt die Struktur der D-Bestimmungen sowie das Procedure-Interface für die Funktion „Cvt2Lower“.

```
.DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++++
 D Cvt2Lower       PI            50A   Varying
 D   ParText                     50A   Varying Options(*Trim) Const
```

Procedure Interface in der klassischen Schreibweise

Wird das Procedure-Interface im Free-Format codiert, so werden die Operations-Codes DCL-PI (Declare Procedure Interface) und END-PI (End Procedure Interface) benötigt.

Der Name der Prozedur wird unmittelbar nach dem DCL-PI Operations-Code angegeben. Sofern die Prozedur einen Rückgabewert hat, wird der Rückgabewert durch die Angabe des Datentyps und der Länge (sofern für den Datentypen erforderlich) definiert. Alternativ ist die Definition über eines der Schlüsselworte LIKE, LIKEDS oder LIKEREC möglich.

Parameter werden nach dem DCL-PI Operations-Code und vor dem END-PI Operations-Code angegeben. Zur Definition der Parameter kann der optionale Operations-Code DCL-PARM verwendet werden. Sofern der Operations-Code DCL-PARM angegeben wird, folgt der Parametername auf den Operations-Code, ansonsten muss der Parametername die erste Angabe sein. Der Parameter wird durch die Angabe des Datentyps und der Länge (sofern für den Datentypen erforderlich) definiert. Alternativ ist auch eine Definition über eines der Schlüsselworte LIKE, LIKEDS oder LIKEREC zulässig.

Das folgende Beispiel zeigt das Procedure-Interface für die Funktion „Cvt2Lower“ im Free-Format.

```
DCL-PI Cvt2Lower VarChar(50);
ParText VarChar(50);
End-Pi;
```

Procedure Interface im Free-Format

3.1.6.1.1 Input-Only-Parameter

Sowohl im fixen als auch im Free-Format können Input-Only-Parameter durch die Angabe eines der folgenden Schlüsselworte definiert werden:

CONST
definiert einen Input-Only-Parameter, der via „Constant Reference“ übergeben wird.

Entspricht der übergebene Parameter der Definition, wird die Adresse des Originalfelds an die aufgerufene Prozedur übergeben.

Wird für den Parameter anstatt des erwarteten Datentyps ein kompatibler Datentyp übergeben, oder weicht die Parameterlänge von der Definition ab, so wird ein temporäres Feld mit der erwarteten Definition erstellt und mit den übergebenen Parameterdaten initialisiert. Anschließend wird die Adresse dieses temporären Felds an die aufgerufene Prozedur übergeben.

Da in der aufgerufenen Prozedur nicht sicher ist, ob die Adresse des Originalfelds oder des temporären Felds übergeben wurde, wurde der Parameter per definitionem als nicht änderbar deklariert.

Änderungen an einem Parameter, der mit „Constant Reference“ übergeben wurde, führen zu Kompilierungsfehlern.

Das Schlüsselwort CONST kann sowohl in Prototypen und Procedure-Interfaces für Prozeduren als auch in Programmen verwendet werden.

3.1.6
Seite 4

VALUE
definiert ebenfalls einen Input-Only-Parameter.

Im Gegensatz zur Parameterübergabe via „Constant Reference“ wird bei der Übernahme „by Value“ immer ein temporäres Feld in der erwarteten Definition generiert und mit den Parameterdaten initialisiert. Dieses temporäre Feld wird dann an die aufgerufene Prozedur übergeben.

Bei sehr großen Parameterfeldern oder Datenstrukturen kann die Übergabe „by Value“ die Performance negativ beeinflussen.

Im Gegensatz zum Schlüsselwort CONST können Parameter, die „by Value“ definiert wurden, in der aufgerufenen Prozedur geändert werden, ohne dass der geänderte Wert an die rufende Prozedur zurückgegeben wird und ohne dass ein Kompilierungsfehler ausgegeben wird.

Das Schlüsselwort VALUE ist nur für Parameter in internen und exportierten Prozeduren erlaubt, nicht jedoch für Programme.

Werden Input-Only-Parameter definiert, können Parameter mit kompatiblen Datentypen abweichender Länge als Ausdruck und sogar als Rückgabewerte aus Built-In-Funktionen und selbstgeschriebenen Funktionen übergeben werden.

Das folgende Beispiel zeigt die Prototypen für die Funktionen „Cvt2Lower“ und „CalcRabatt“. Alle Parameter sind als Input-Only-Parameter definiert. An die Funktion „Cvt2Lower“ wird der Parameter via „Constant Reference“ übergeben (1), während die Parameterübergabe an die Funktion „CalcRabatt“ „by Value“ (2) erfolgt.

Im Anschluss werden beide Funktionen mehrfach aufgerufen.

Die Variable, die beim ersten Aufruf der Funktion „Cvt2Lower“ verwendet wird ist kürzer, als die laut Parameterdefinition erwartete Variable (3).

Beim zweiten Aufruf der Funktion „Cvt2Lower“ wird ein Ausdruck, der aus zwei Variablen in Verbindung mit der Built-In-Function %TRIM gebildet wird, übergeben (4).

Beim dritten Aufruf der Funktion „Cvt2Lower“ wird ein Text direkt übergeben (5).

Beim ersten Aufruf der Funktion „CalcRabatt“ werden zwei Variablen übergeben, die jedoch beide von der erwarteten Definition abweichen (6).

Beim zweiten Aufruf der Funktion „CalcRabatt“ wird der erste Parameterwert aus den Variablen Menge und Preis ermittelt (7).

Beim dritten Aufruf der Funktion „CalcRabatt“ wird der erste Parameter wiederum aus Menge und Preis ermittelt, das Ergebnis jedoch auf zwei Nachkommastellen kaufmännisch gerundet. Der zweite Parameter wird fix mit 2,5 übergeben (8).

Beim letzten Aufruf der Funktion „CalcRabatt“ wird zur Berechnung des ersten Parameters eine weitere individuelle Funktion aufgerufen und deren Rückgabewert an die Funktion „CalcRabatt“ übergeben (9).

```
 D Cvt2Lower       PR             50A   Varying
 D  ParText                      50A   Varying Const

(1)
 D CalcRabatt      PR             13P 2 Varying
 D  ParWert                      11P 2 Value
 D  ParProzent                    5P 2 Value
(2)

(2)
 D Text            S              20A
 D Wert            S               7P 2
 D Prozent         S               3U 0
 D Ergebnis        S              11P 2
 D NeuerText       S             256A   Varying
  *---------------------------------------------------------
  /Free
    NeuerText = Cvt2Lower(Text);
(3)
    NeuerText = Cvt2Lower(%Trim(Nachname) + ', ' +
%Trim(Vorname);  (4)
    NeuerText = Cvt2Lower('Dieser Text wird konvertiert');
(5)

    Ergebnis  = CalcRabatt(Wert: Prozent);
(6)
    Ergebnis  = CalcRabatt(Menge * Preis: Prozent);
(7)
    Ergebnis  = CalcRabatt(%DecH(Menge * Preis: 11: 2): 2,5);
(8)
    Ergebnis  = CalcRabatt(CalcGesamtWert(Menge: Preis):
Prozent);  (9)
  /End-Free
```

Aufruf von Funktionen mit Input-Only-Parameter

3.1.6.1.2 Parameteroptionen

In Verbindung mit dem Schlüsselwort OPTIONS können weitere Parameteroptionen angegeben werden. Das OPTIONS-Schlüsselwort kann pro Parameter zwar nur einmal angegeben werden, pro OPTIONS-Schlüsselwort können jedoch mehrere Ausprägungen – durch Doppelpunkt getrennt – angegeben werden, zum Beispiel OPTIONS(*NoPass: *Trim). Bei dem Schlüsselwort OPTIONS sind die folgenden Ausprägungen zulässig:

*NOPASS
Durch die Ausprägung *NOPASS kann ein optionaler Parameter definiert werden.

Alle folgenden Parameter müssen ebenfalls mit dem Schlüsselwort OPTIONS(*NOPASS) definiert werden.

*OMIT
Anstatt des Parameters kann über den Sonderwert *OMIT ein NULL-Pointer übergeben werden.

Die Ausprägung ist nur für Parameter, die mit „(Constant) Reference“ übergeben werden.

*TRIM
Kann nur bei alphanumerischen Input-Only-Parametern verwendet werden. Bei Parametern mit fixer Länge werden die Inhalte linksbündig ausgerichtet. Bei Parametern mit variabler Länge werden führende und folgende *Blanks entfernt.

*RIGHTADJ
In alphanumerischen Input-Only-Parametern mit fixer Länge wird der Parameterwert rechtsbündig ausgerichtet.

*STRING
erlaubt bei Pointer-Parametern die wahlweise Übergabe eines Pointers oder eines alphanumerischen Strings.

*VARSIZE
erlaubt die Übergabe von alphanumerischen oder Unicode-Ein-/Ausgabeparametern mit unbestimmter Länge.

In der aufgerufenen Prozedur muss dafür gesorgt werden, dass nur genau die Anzahl an übergebenen Bytes verarbeitet wird.In der Regel wird die Länge des übergebenen Parameters in einem weiteren Parameter übergeben.

Achtung:

Die Option OPTIONS(*VARSIZE) muss vorsichtig gehandhabt werden, da falsche Längenberechnungen oder -angaben dazu führen, dass globaler Speicher überschrieben wird.

Solche Fehler treten meist erst viel später und an einer völlig anderen und unerwarteten Stelle auf. Die Ursache dieser Fehler ist sehr schwer zu finden und zu beheben.

Die Parameter müssen in der Reihenfolge übergeben werden, in der sie definiert wurden.

3.1.6.2 Prototyp

Prototypen werden zwecks Parameterprüfung zur Compile-Zeit benötigt.

In der Quelle, in der die Main-Procedure für ein Programm oder eine exportierte Procdure hinterlegt wird, muss ein Prototyp hinterlegt werden, der der Parameterbeschreibung entspricht. Bei der Kompilierung wird die Parameterdefinition mit dem Prototypen abgeglichen. Stimmen Prototyp und Procedure Interface nicht überein, so wird ein Kompilierungsfehler ausgegeben.

Seit Release 7.1 ist die Angabe der Prototypen für interne Prozeduren optional, das heißt, der Abgleich erfolgt direkt zwischen Aufruf und Procedure Interface.

Um eine Parameterprüfung zur Compile-Zeit zu ermöglichen, muss der Prototyp für eine Prozedur oder ein Programm in jede Quelle eingebunden werden, in der die Prozedur aufgerufen wird. Beim Kompilieren werden die übergebenen Parameter mit dem Prototypen abgeglichen. Stimmen Prototyp und Parameterdefinition nicht überein, so wird ein Kompilierungsfehler ausgegeben.

Anmerkung:

Um sicherzustellen, dass die Prototypen auch wirklich der Parameterbeschreibung entsprechen, sollte der Prototyp nur einmalig in einer separaten Teildatei hinterlegt werden. Diese Teildatei sollte in allen Teildateien, in denen die entsprechende Funktion aufgerufen wird, hinterlegt werden.

Niemals Prototypen hart in den Quellen hinterlegen!

3.1.6
Seite 8

Wird der Prototyp im fixen Format codiert, so entspricht er dem Procedure-Interface. Anstatt auf den Positionen 24–25 PI zu hinterlegen, wird beim Prototypen PR angegeben. Gleiche Parameternamen in verschiedenen Prototypen führen sogar mit unterschiedlicher Definition zu keinem Fehler. Vielmehr sind Parameternamen in Prototypen optional, das heißt, für den Abgleich werden lediglich Anzahl, Art und Reihenfolge der Parameter berücksichtigt.

Das folgende Beispiel zeigt den Prototypen für die Funktion „Cvt2Upper“. In diesem Fall ist der einzige Unterschied zwischen Procedure Interface und Prototyen in der ersten Zeile zu finden. Beim Prototypen wird in Position 24-25 PR (=Prototyp) angegeben, während beim Procedure Interface die Angabe von PI (=Procedure Interface) erforderlich ist.

```
.DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++++(
D Cvt2Lower       PR                  50A   Varying
D  ParText                            50A   Varying Options(*Trim) Const
```

Prototyp in klassischer Schreibweise

Wird der Prototyp im Free-Format codiert, müssen die Operations-Codes DCL-PR (Declare Prototype) und END-PR (End Prototype) verwendet werden.

Ansonsten entspricht der Aufbau des Prototypen dem Procedure Interface. Ein Rückgabewert wird direkt nach dem Prozedurnamen definiert. Die einzelnen Parameter werden zwischen DCL-PR und END-PR hinterlegt. Parameternamen in Prototypen werden zwar nicht geprüft, im Gegensatz zum fixen Format muss jedoch ein Parametername, zumindest aber der Sonderwert *N angegeben werden.

Das folgende Beispiel zeigt das Procedure-Interface für die Funktion „Cvt2Lower“ im Free-Format. Anstatt des Parameternamens wird der Sonderwert *N (1) verwendet.

```
DCL-PR Cvt2Upper VarChar(50);
*N VarChar(50) Options(*Trim) const; (1)
End-PR;
```

Prototyp im Free-Format

3.1.6.2.1 Prototypen für Programme und Prozeduren

Prototyping wurde ursprünglich für den Aufruf von Prozeduren eingeführt. Der Prozedurname wird nach RPG-Manier intern in Großbuchstaben konvertiert.

Die ILE-Konzepte gehen jedoch davon aus, dass Prozeduren und Funktionen, die in anderen Programmiersprachen geschrieben sind, aufgerufen werden können. Um den Aufruf solcher Funktionen/Methoden zu ermöglichen, muss ein Prototyp angelegt und eingebunden werden.

Da Prozedurnamen in anderen Sprachen case sensitiv sein können, kann der „echte" Prozedur-/Funktionsname in Verbindung mit dem Schlüsselwort EXTPROC, das in der PR-Zeile angegeben wird, hinterlegt werden.

Das folgende Beispiel zeigt den Prototyp des unix-type API „read". Bei den unix-type APIs handelt es sich i.d.R. um C-Funktionen. Der korrekte Funktionsname (in Kleinbuchstaben) wird im Schlüsselwort EXTPROC angegeben.

```
D read            PR            10I 0 ExtProc('read')
(1)
D  fildes                       10i 0 value
D  buf                            *   value Options(*String)
D  bytes                        10U 0 value
```

Prototyp für das unix-type API „read"

Wie bereits erwähnt, wurde Prototyping ursprünglich nur für Prozeduren eingeführt und erst später auch für Programme zugelassen.

Für den Aufruf von Programmen und Prozeduren wurde nur ein einziger Operations-Code bereitgestellt, nämlich CALLP (Call with Prototype). Die Unterscheidung, ob ein dynamischer Programm- oder ein statisch gebundener Prozeduraufruf erfolgen soll, wird über ein Schlüsselwort in der PR-Zeile des Prototyps gesteuert.

Für Programme muss das Schlüsselwort EXTPGM angegeben werden. Fehlt dieses Schlüsselwort oder wurde es nicht angegeben, so wird davon ausgegangen, dass es sich um eine Prozedur handelt.

Bei einem dynamischen Programmaufruf erfolgt zur Compile-Zeit keine Prüfung, ob das Objekt/Programm bereits existiert.

Bei einem Prozeduraufruf muss entweder das Modul oder die Signatur des Serviceprogramms, in dem sich die Prozedur befindet, ermittelt und eingebunden werden. Aus diesem Grund müssen Prozeduren vor der Kompilierung des rufenden Programms oder der rufenden Prozedur vorhanden sein.

Ergänzung 29./2014

3.1.6
Seite 10

Der Aufrufname auf den Positionen 7–21 kann von dem eigentlichen Programm- oder Prozedurnamen abweichen. Damit können kryptische Programmnamen in sprechende Namen verwandelt werden.

Anmerkung:

Durch einen abweichenden Aufrufnamen kann die Lesbarkeit der Programme erheblich verbessert werden.

Durch sprechende Aufrufnamen kann bereits eine Dokumentation in der Codierung erreicht werden.

Das folgende Beispiel zeigt den Prototypen für das Programm XY0401R (1) sowie den Aufruf des Programms. Um den Source Code einfacher lesbar zu machen und quasi zu dokumentieren, wurde der Aufrufname „WartungAuftragskopf“ festgelegt.

Der Aufruf erfolgt für Mandant 100 über den Aufrufnamen. (2)

```
 D WartungAuftragsKopf...
 D                 PR                  EXTPGM('XY0401R')
(1)
 D  ParMandant                    3P 0 Const

  /Free
    WartungAuftragsKopf(100);
(2)
  /End-Free
```

Prototyp für Programm XY0401R und Aufruf des Programms

3.1.7 ILE-Objekte

Mit den ILE-Konzepten wurde auch eine Reihe von neuen Objekten bzw. Objektarten eingeführt:

- Module
  bilden die Basis für Programme und Serviceprogramme.
- Programme
  sind zwar nicht neu, können sich jetzt jedoch aus mehreren Modulen zusammensetzen, die in unterschiedlichen Programmiersprachen geschrieben wurden.
- Serviceprogramme
  ermöglichen eine (hoch-)modulare Programmierung mit Einsatz von wiederverwendbarem Source Code ohne Duplizierung.
- Binderverzeichnisse
  erleichtern das Erstellen von Programmen und Service-Programmen in (hoch-)modularen Anwendungen.

3.1.7
Seite 2

3.1.7.1 Module – Basis für Programme und Serviceprogramme

Integrated Language Environment bedeutet nicht nur, dass Prozeduren und Programme, die in anderen Sprachen geschrieben wurden, aufgerufen werden können, sondern auch, dass ein Programm aus Teilen bestehen kann, die in verschiedenen Sprachen geschrieben wurden.

Um ein solches Programm generieren zu können, muss zunächst der Compiler den Source Code in Maschinencode übersetzen. Da jede Programmiersprache ihren eigenen Compiler hat, müssen die Einzelbausteine folglich mit Hilfe des entsprechenden Language Compilers erstellt werden. Das Ergebnis aus einer solchen Kompilierung ist ein Objekt mit der Objektart *MODULE.

Module werden durch Aufruf des entsprechenden CL-Befehls CRTXXXMOD (Modul erstellen) generiert, wobei XXX als Platzhalter für die Programmiersprache steht.

Sofern in den Quellen embedded SQL enthalten ist, wird das Modul mit dem Befehl CRTSQLXXXI (SQL ILE-Objekt erstellen) erstellt, wobei XXX die Programmiersprache repräsentiert.

Nach Überarbeitung der Quelle durch den SQL-Pre-Compiler wird der entsprechende Language Compiler erstellt, das heißt, das Programm wird durch Ausführung des entsprechenden CRTXXXMOD-Befehls generiert.

Anmerkung:

Zur Erstellung von Programmen und Modulen, die embedded SQL beinhalten, wird in jeder Programmiersprache nur ein einziger Befehl zur Verfügung gestellt (CRTSQLXXXI). Ob ein Programm oder ein Modul erstellt wird, hängt davon ab, welcher Objekttyp bei der Umwandlung angegeben wurde.

Ein Modul beinhaltet zwar umgewandelten Source Code, ist jedoch nicht direkt ausführbar. Weder das Modul noch die darin enthaltenen Prozeduren können direkt (zum Beispiel von einer Befehlszeile aus) aufgerufen werden. Erst nachdem die Module zu Programmen oder Serviceprogrammen gebunden wurden, können die in den Modulen integrierten Prozeduren ausgeführt werden.

Das Binden von Modulen zu Programmen oder Serviceprogrammen erfolgt im Anschluss mit Hilfe der Befehle CRTPGM (Programm erstellen) oder CRTSRVPGM (Serviceprogramm erstellen).

Module werden nach dem Einbinden in Programme und Serviceprogramme nicht mehr benötigt. Sowohl in die Programme als auch in die Serviceprogramme werden Duplikate der Module übernommen. Diese eingebundenen Duplikate werden zur Laufzeit ausgeführt.

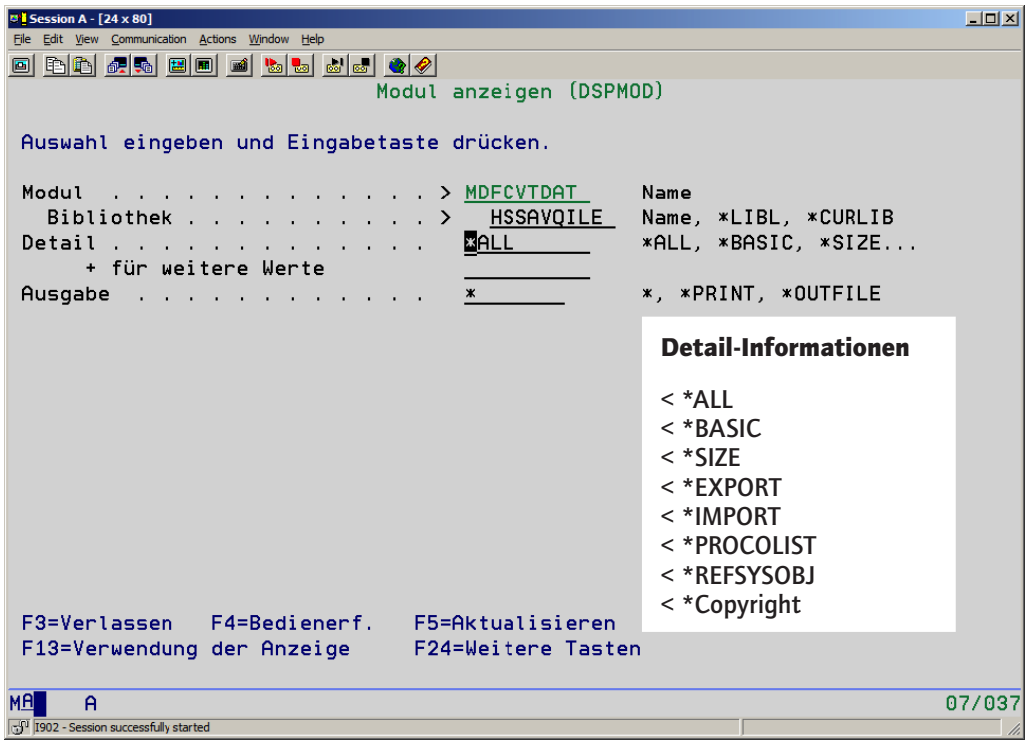
Anmerkung:

Nach dem Binden werden die Modulobjekte nicht mehr benötigt und können bzw. sollten gelöscht werden.

3.1.7.1.1 CL-Befehl DSPMOD – Modulinformationen anzeigen

Mit dem Befehl DSPMOD (Modulinformationen anzeigen) können eine Vielzahl von Informationen über das Modul und dessen Erstellung angezeigt werden.

Die folgende Abbildung zeigt die erste Seite des Befehls DSPMOD (Modul anzeigen).



CL-Befehl DSPMOD – Modulinformationen anzeigen

Anmerkung:

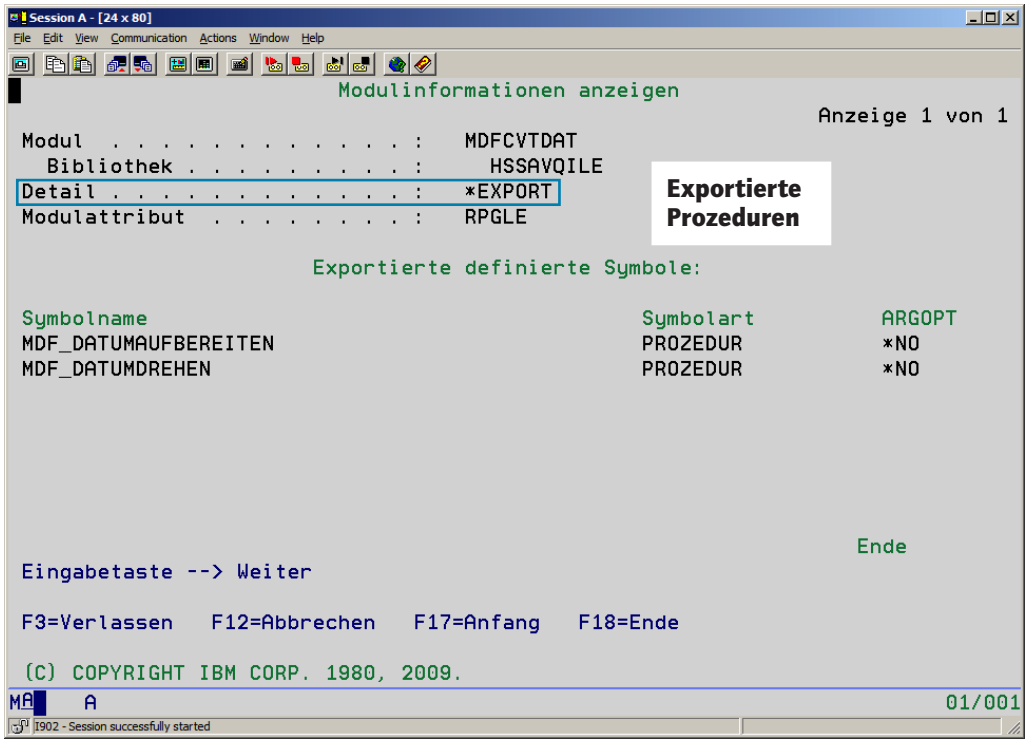
Nur über den Befehl DSPMOD ist es möglich, festzustellen, aus welcher Quelle ein Modul erstellt wurde.

Das gilt auch dann, wenn ein Programm in einem einzigen Schritt mit dem Befehl CRTBNDxxx (Binder-Programm erstellen – xxx = Programmiersprache) generiert wurde. Beim Erstellen eines Binder-Programms wird zunächst aus der Quelle ein Modul erstellt, das im Anschluss in ein Programm gebunden wird. Nach erfolgreicher Erstellung des Programms wird das Modul wieder gelöscht.

Insbesondere die Detailoptionen *EXPORT und *PROCLIST enthalten wichtige Informationen.

- *EXOPRT
  Die Auswahl *EXPORT liefert eine Übersicht über alle aus dem Modul exportierte Prozeduren, also über alle Prozeduren, die von außerhalb des Moduls aufgerufen werden können, das heißt aus anderen Programmen oder Prozeduren in anderen Serviceprogrammen.
- *PROCLIST
  Bei der Auswahl *PROCLIST werden alle im Modul definierten internen und aus dem Modul exportierten Prozeduren aufgelistet.

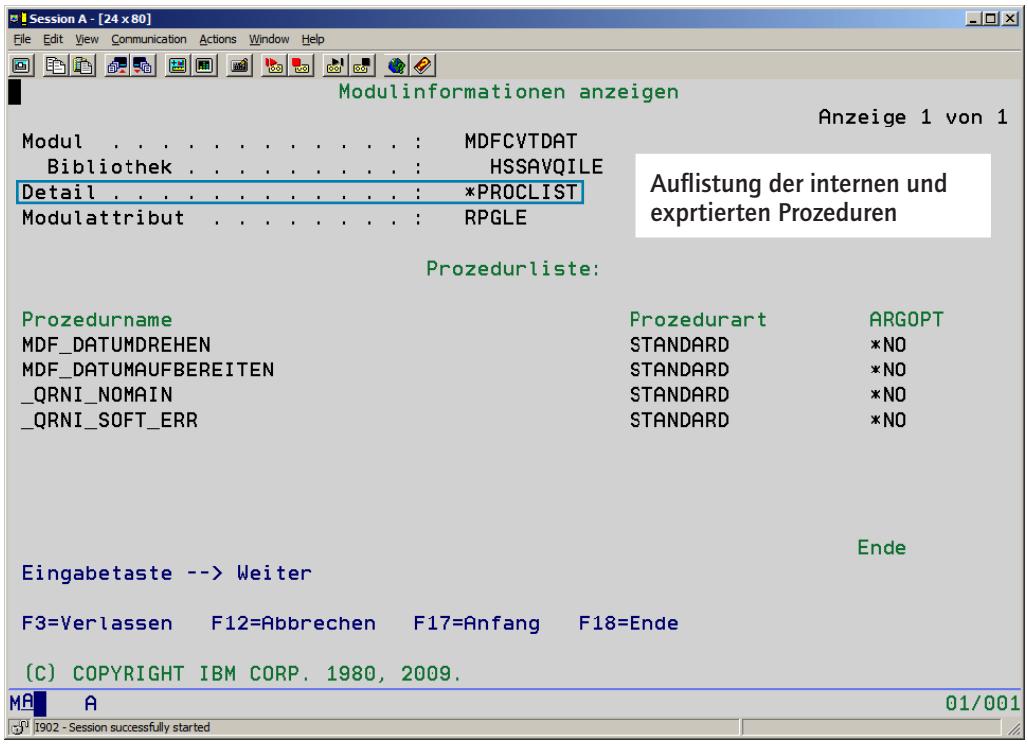
Die folgende Abbildung zeigt die aus dem Modul MDVCVTDAT exportieren, also von außerhalb des Moduls aufrufbaren Prozeduren. Im vorliegenden Beispiel können die Prozeduren MDF_DATUMAUFBEREITEN und MDF_DATUMDREHEN von anderen Prozeduren in anderen Programmen oder Serviceprogrammen aufgerufen werden.



*DSPMOD – Option *EXPORT = Exportierte Prozeduren*

3.1.7
Seite 6

Die folgende Abbildung zeigt die Detailinformation *PROCLIST. Bei dieser Auswahl werden alle internen und exportierten Prozeduren, die innerhalb des Moduls definiert wurden, ausgegeben. Im vorliegenden Fall sind die individuellen (exportierten) Prozeduren MDF_DATUMDREHEN und MDF_DATUMAUFBEREITEN im Modul hinterlegt.



*DSPMOD – Option *PROCLIST – Auflistung aller Prozeduren*

3.1.7.2 Programme

Programme sind ausführbare Objekte der Objektart *PGM, die über einen CALL-Befehl zum Beispiel von einer Befehlszeile aus oder aus einem Menü heraus aufgerufen werden können.

Programme werden dynamisch aufgerufen, das heißt, erst in dem Moment, in dem ein Programm aufgerufen wird, wird geprüft, ob es existiert. Beim Aufruf eines Programms wird das Programm in den Speicher geladen, die verwendeten Dateien geöffnet, Speicher für die diversen Variablen reserviert und initialisiert etc. Das Programm wird in Laufbereitschaft versetzt, das heißt, es wird aktiviert. Nach der Aktivierung wird die Kontrolle an die Programm-Eingangs-Prozedur (PEP = Program Entry Procedure) übergeben.

Ein ILE-Programm kann sich aus mehreren kopierten Modulen zusammensetzen. Die Module können entweder in der gleichen oder in verschiedenen Sprachen geschrieben worden sein. Mindestens eines der Module benötigt jedoch eine Main-Procedure, da nur Main-Procedures direkt aufgerufen werden können.

Main Procedure

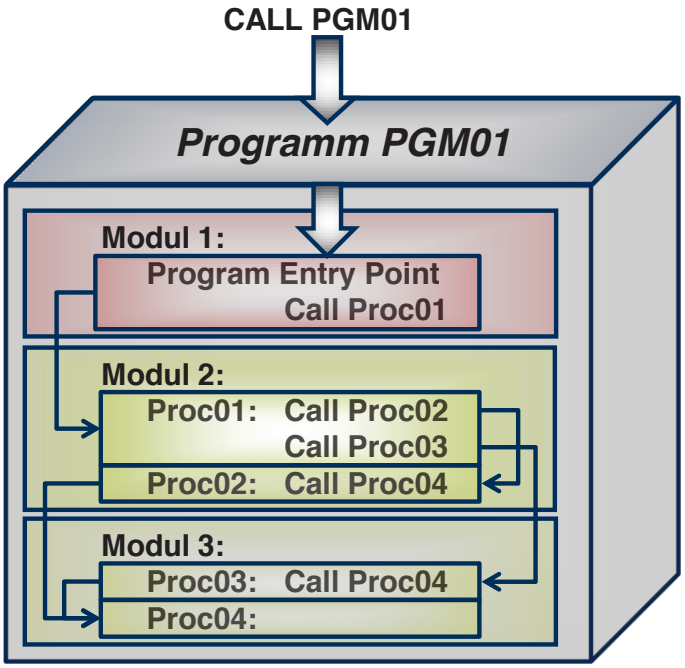
Unter einer (zyklischen) Main-Procedure werden alle Statements verstanden, die vor der ersten internen oder exportierten Prozedur codiert wurden.

Sofern das Modul keine weiteren Prozeduren enthält, entspricht die Main-Procedure der Codierung eines klassischen OPM (Original Program Model-)Programms (zum Beispiel RPGIII-Programm).

Sofern mehrere Module eine Main-Procedure haben, muss bei der Programmerstellung, das heißt beim Binden der Module, der Programm-Eingangs-Punkt/Program-Entry-Point (PEP) also die aufzurufende Main-Procedure angegeben werden.

3.1.7
Seite 8

Das folgende Beispiel zeigt schematisch den Aufbau des Programms PGM01. In diesem Programm sind die Module „Modul1“, „Modul2“ und „Modul3“ gebunden. Modul 1 beinhaltet eine Main-Procedure und wurde als Program-Entry-Point (PEP-Modul) deklariert. Aus Modul 1 wird die Prozedur Proc01, die sich im Modul 2 befindet, aufgerufen. Modul 2 beinhaltet die Prozeduren Proc01 und Proc02. Aus diesen Prozeduren werden weitere Prozeduren, die sich in Modul 2 und Modul 3 befinden, aufgerufen. PGM01 beinhaltet somit alle Prozeduren, die in diesem Programm ausgeführt werden.



ILE-Programm mit drei Modulen

3.1.7.2.1 Erstellen von ILE-Programmen

Solange das Programm aus einer einzigen Teildatei erzeugt wird, kann die Kompilierung einfach mit Hilfe des Befehls CRTBNDXXX (Binder-Programm erstellen – XXX = Programmiersprache) erfolgen.

Sofern die Teildatei embedded SQL beinhaltet, kann die Kompilierung durch den Befehl CRTSQLXXXI (SQL ILE-Objekt erstellen) unter Angabe des Objekttyps *PGM erfolgen. Nach der Verarbeitung durch den SQL-Pre-Compiler wird der sprachenabhängige CRTBNDXXX-Befehl (Binder-Programm erstellen) ausgeführt.

Im Gegensatz zur Erstellung von OPM-Programmen ist das Erstellen von ILE-Programmen jedoch immer ein zweistufiger Prozess:

- Erstellen aller Module
- Binden der Module in ein Programm

Dieser zweitstufige Prozess findet auch dann statt, wenn das Programm mit Hilfe des Befehls CRTBNDxxx (Binder-Programm erstellen) aus einer einzigen Teildatei generiert wird. In diesem Fall wird das Modul in der Bibliothek QTEMP erstellt und anschließend in ein Programmobjekt in der Objektbibliothek kopiert. Nach erfolgreicher Erstellung des Programms wird das Modul wieder aus der Bibliothek QTEMP entfernt.

3.1.7.2.1.1 Bind by Copy – Module in Programme einbinden

Sofern mehrere Module zu einem Programm gebunden werden sollen, müssen zuvor alle Module durch den Language Compiler mit den entsprechenden Kompilierungsbefehlen CRTXXXMOD (Modul erstellen) erzeugt worden sein.

Das Erstellen von Programmen bzw. das Binden von Modulen zu Programmen kann auf zwei Arten erfolgen:

- Befehl CRTPGM
  Programm erstellen
  Die einzelnen Module werden aufgelistet.
- Befehl CRTBNDXXX
  Binder-Programm erstellen
  Die einzelnen Module sind in einem Binder-Verzeichnis eingetragen. Anstatt die Module einzeln aufzulisten, wird nur das Binder-Verzeichnis angegeben.
  Die Module, aus denen Prozeduren aufgerufen werden, werden über das Binder-Verzeichnis ermittelt und anschließend in das Programmobjekt kopiert.

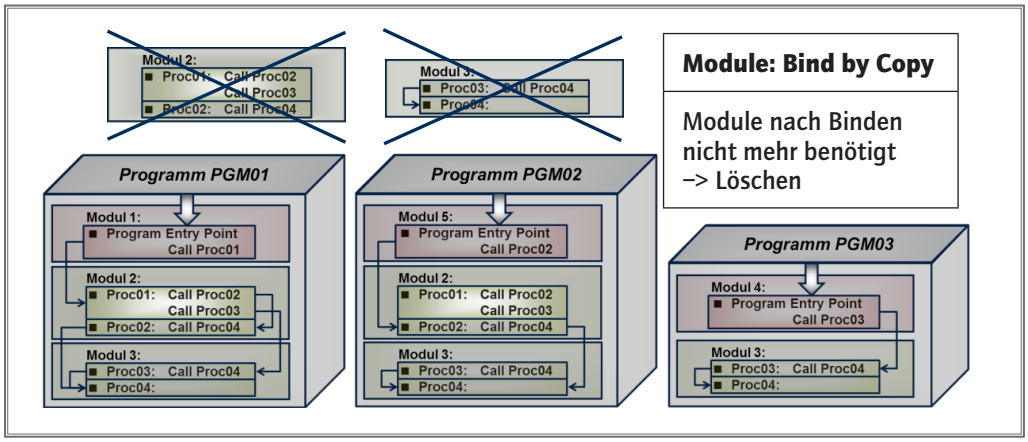
Beim Binden der Module – unabhängig davon, welcher Befehl verwendet wird – werden Duplikate der Module in die Programmobjekte integriert (Bind by Copy). Wird das gleiche Modul in mehrere Programme und/oder Serviceprogramme gebunden, so wird ein Duplikat des Moduls in jedes dieser Objekte eingebunden.

Anmerkung:

Die Modulobjekte werden zur Laufzeit nicht benötigt, da jedes Programm ein Duplikat des Moduls enthält, das zur Laufzeit ausgeführt wird.

Nach dem Binden sind die Modulobjekte somit nicht länger erforderlich und können bzw. sollten gelöscht werden.

Das folgende Schaubild zeigt drei Programme, in die jeweils die Module „Modul2“ und „Modul3“ by Copy eingebunden wurden. Nach dem Binden werden die Module gelöscht.



Bind by Copy – Programme mit integrierten Modulen

Da beim „Bind by Copy“ ein Duplikat des Moduls in jedes Programm eingebunden wurde, muss nach Änderung des Moduls (zum Beispiel nach einer Fehlerbehebung) das Modul in jedem dieser Programme ausgetauscht werden.

Der Austausch von Modulen kann über den Befehl UPDPGM (Programm aktualisieren) erfolgen. Alternativ können auch alle Module neu erstellt und die Programme neu gebunden werden.

Da Module by Copy gebunden und bei Änderung alle Modulkopien ausgetauscht werden, sollten Module nur in Ausnahmefällen mehrfach gebunden werden.

Anmerkung:

Ändert sich ein Modul, muss es in allen Programmen, in denen es eingebunden ist, ausgetauscht werden, das heißt, die Programme müssen neu gebunden/erstellt werden.

Da Module by Copy gebunden werden, sollte ein Modul nur einmal eingebunden werden – entweder in ein Programm oder in ein Serviceprogramm.

In (hoch)modularen Anwendungen werden die Module mit universell einsetzbaren Prozeduren in Serviceprogramme gebunden.

3.1.7.2.1.2 CRTPGM – Programm erstellen

Module können mit Hilfe des Befehls CRTPGM zu einem Programm gebunden werden.

Dabei muss zunächst der Programmname angegeben werden. Der Programmname kann völlig frei gewählt werden, unabhängig vom Namen der zu bindenden Modulobjekte.

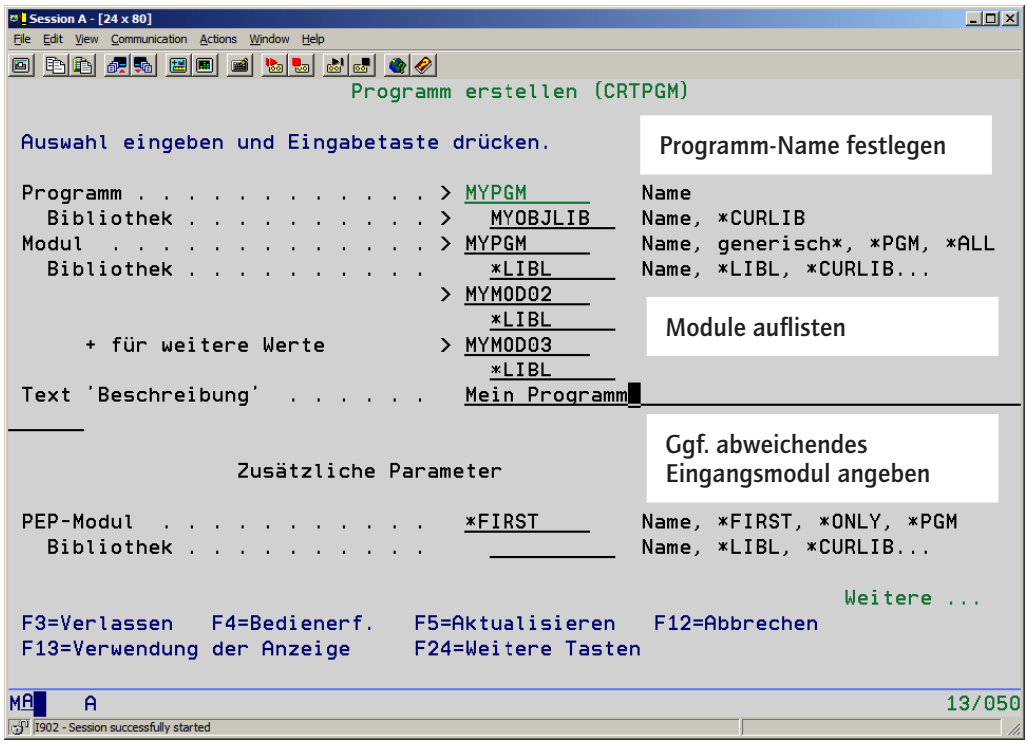
Tipp:

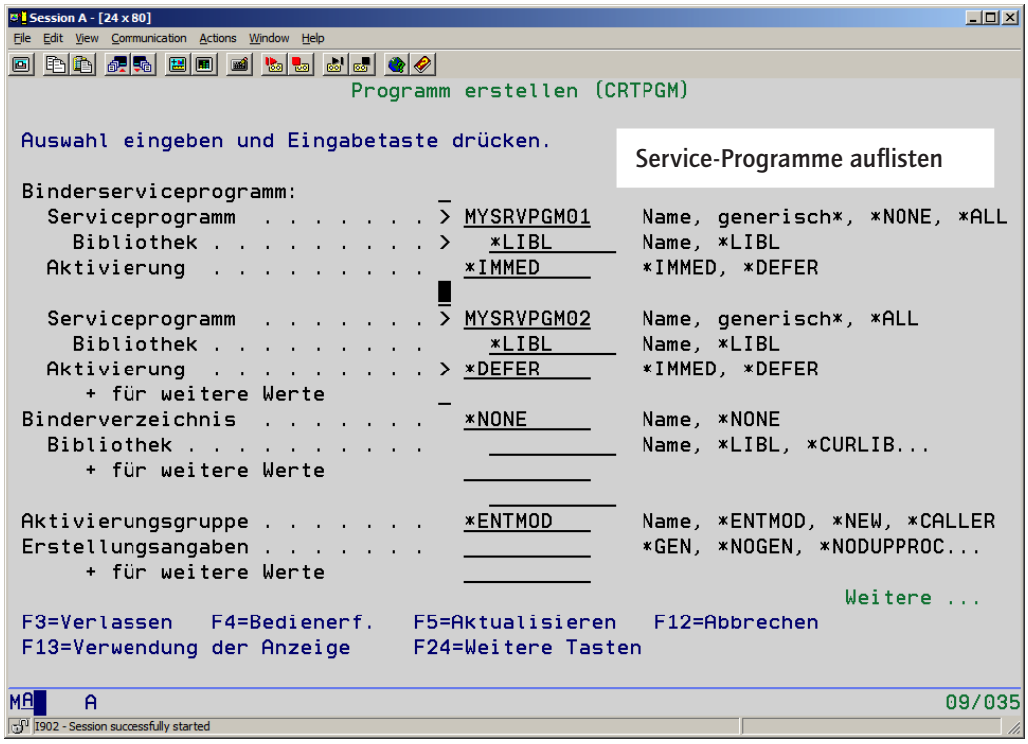
Auch wenn der Programmname unabhängig vom Namen der zu bindenden Module gewählt werden kann, hat es sich als vorteilhaft erwiesen, für das Programm den gleichen Namen wie für das PEP-Modul zu verwenden.

Die zu bindenden Module müssen aufgelistet werden. Das Eingangsmodul, das heißt der Programm-Entry-Point, ist zu benennen.

Sofern aus den zu bindenden Modulen Prozeduren aufgerufen werden, die in Serviceprogrammen hinterlegt sind, sind diese Serviceprogramme ebenfalls aufzulisten.

Die folgende Abbildung zeigt den Befehl CRTPGM (Programm erstellen):





Befehl CRTPGM – Programm erstellen

3.1.7
Seite 14

Man kann sich leicht vorstellen, dass die Erstellung eines Programms bei einer sehr modularen Programmierung zu einer aufwendigen Angelegenheit werden kann.

Um hier für Vereinfachung zu sorgen, sind einige Unternehmen dazu übergegangen, für die Kompilierung CL-Programme zu erstellen, die zunächst die Module und im Anschluss daran die Programme generieren. Ein aufwendiges und fehlerträchtiges Prozedere, da für jedes neue Modul und/oder Programm die CL-Programme angepasst werden müssen.

In anderen Unternehmen werden die Kompilierungsbefehle als Kommentar im Header der Quellen hinterlegt. Beim Erstellen werden dann diese Kommentare ausgelesen und mit Hilfe des APIs (Application Programming Interface) QCMDEXC (CL-Befehl ausführen) ausgeführt. Auch hier besteht die Gefahr, dass zwar die Quellen korrekt geändert, die als Kommentare hinterlegten Compile-Befehle jedoch nicht angepasst wurden.

Einfacher ist es, die Module und/oder Serviceprogramme in Binder-Verzeichnissen einzutragen und diese Verzeichnisse beim Binder-Schritt (zum Beispiel bei Ausführung des Befehls CRTPGM) anzugeben. Ein Binder-Verzeichnis für alle Module, die direkt gebunden werden, und ein Binder-Verzeichnis, in dem alle Serviceprogramme hinterlegt sind, sollten für diesen Zweck ausreichen.

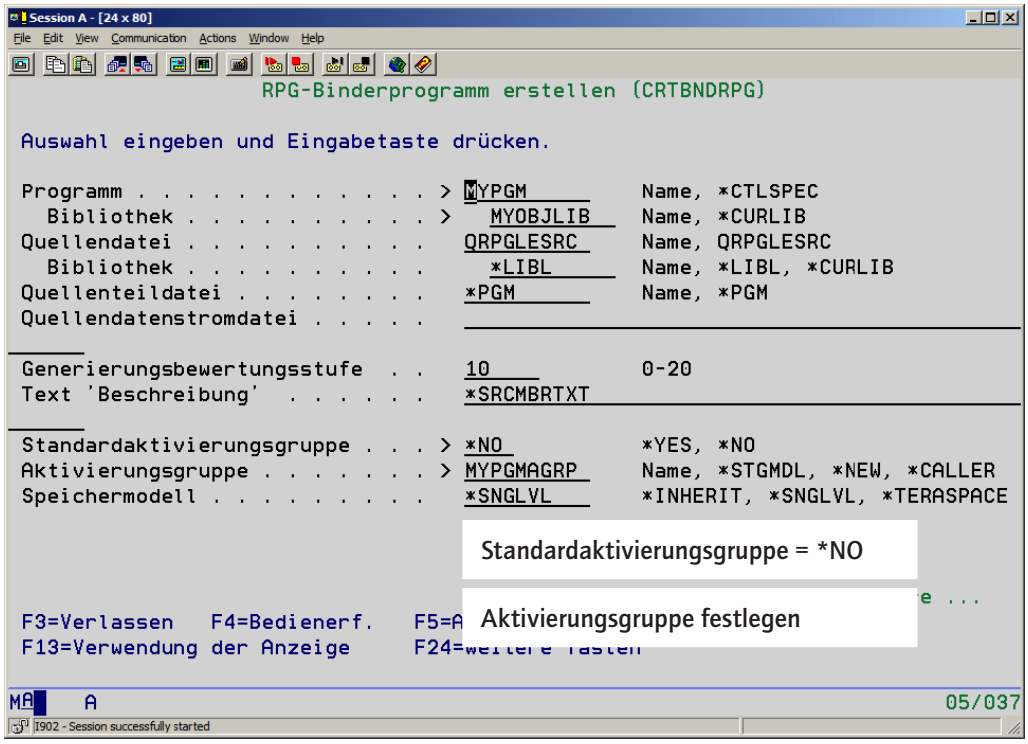
Im Binder-Schritt werden die Binder-Verzeichnisse durchsucht. Sofern die aufgerufenen Prozeduren in einem Modul oder Serviceprogramm, das im Binder-Verzeichnis eingetragen ist, gefunden werden, wird entweder das Modul oder die Signatur des Serviceprogramms in das Programmobjekt übernommen. Sofern eine Prozedur weder in einem Modul noch in einem Serviceprogramm gefunden wird, kann das Programm nicht erstellt werden.

Die Binder-Verzeichnisse können im CRTPGM-Befehl aufgelistet werden. Eine Auflistung der Module und Serviceprogramme ist dann nicht mehr erforderlich.

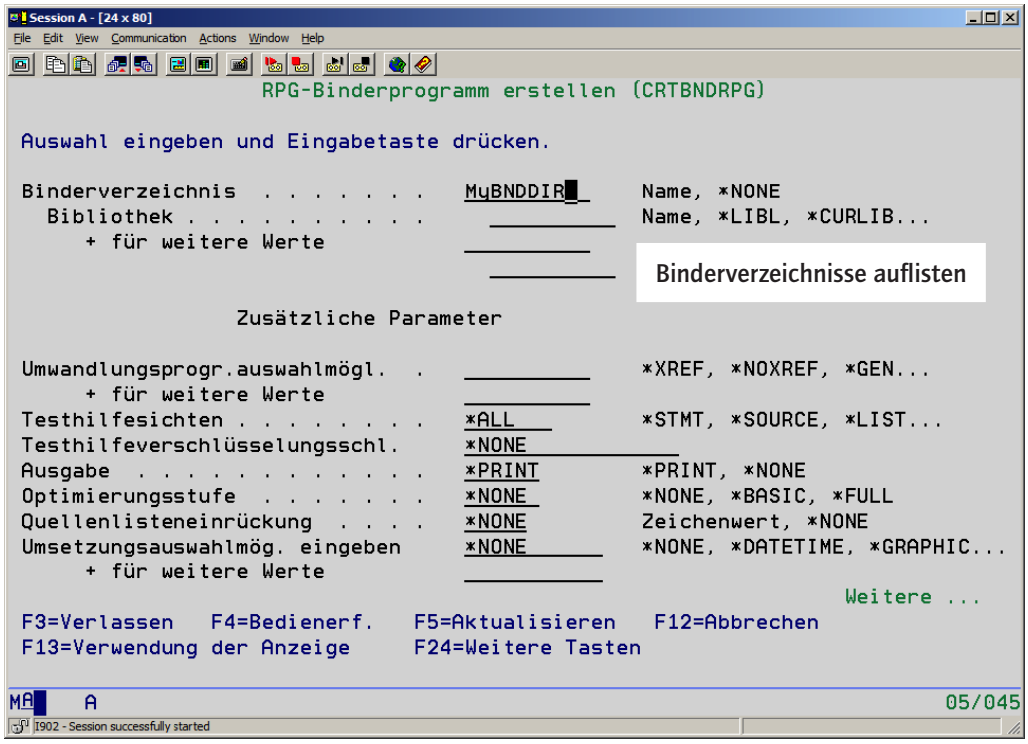
3.1.7.2.1.3 CRTBNDXXX – Binder-Programm erstellen

Werden die Module und Serviceprogramme in ein Binder-Verzeichnis eingetragen, so kann auch die Erstellung des Programms zusammen mit der Erstellung des PEP-Moduls direkt über den Befehl CRTBNDXXX erfolgen.

Dabei muss die Standardaktivierungsgruppe auf *NO geändert und eine Aktivierungsgruppe vorgegeben werden. Nach Änderung der Standardaktivierungsgruppe ist es möglich, ein oder mehrere Binder-Verzeichnisse anzugeben, die während des Binder-Schritts durchsucht werden.



3.1.7
Seite 16



Befehl CRTBNDRPG – Binder-Programm mit Binder-Verzeichnis erstellen

Zur Umwandlung mit CRTBNDRPG können die Aktivierungsgruppe und die Binder-Verzeichnisse in den H-Bestimmungen des Programmcodes hinterlegt werden.

Das folgende Beispiel zeigt die notwendigen H-Bestimmungen zunächst in der fixen und anschließend in der Format-Codierung. Die Schlüsselworte DFTACTGRP und ACTGRP sind durch eine Compiler Directive bedingt, das heißt, sie werden nur berücksichtigt, wenn das Programm mit dem Befehl CRTBNDRPG (Binder-Programm erstellen) umgewandelt wird. Wird dagegen die Quelle mit dem Befehl CRTRPGMOD (Modul erstellen) kompiliert, werden diese Einträge in den H-Bestimmungen ignoriert. Die Binder-Verzeichnisse werden immer eingebunden.

Seit Release 7.1 TR 7 ist die Angabe des Schlüsselworts DFTACTGRP(*NO) optional. Wird eines der Schlüsselworte ACTGRP, BNDDIR oder STGMDL in den H-Bestimmungen angegeben, wird DFTACTGRP(*NO) durch den RPG-Compiler automatisch gesetzt.

```
/If Defined (*CRTBNDRPG)
H DFTACTGRP(*NO) ACTGRP('MYPGM')
 /EndIf
H BNDDIR('MODBNDDIR': 'SRVBNDDIR')
```

```
   CTL-Opt
     /If Defined (*CRTBNDPRG)
      ActGrp('MYPGM')
     /EndIf
      Bnddir('MODBNDDIR': 'SRVBNDDIR');
```

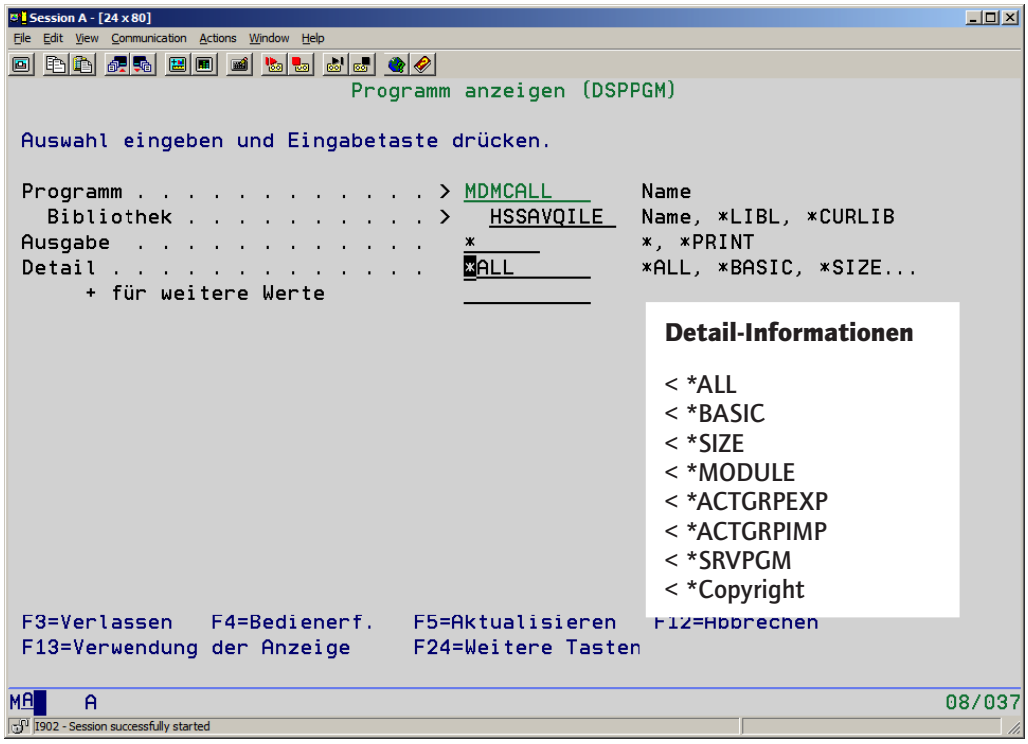
H-Bestimmungen – Aktivierungsgruppe und Binder-Verzeichnis

Sofern die Module nach dem Löschen nicht entfernt werden, oder wenn alle Module mit universell einsetzbaren Prozeduren in Serviceprogrammen gebunden sind, und diese Module und Serviceprogramme in Binder-Verzeichnissen hinterlegt sind, kann jedes ILERPG-Programm mit einem einzigen CRTBNDRPG-Befehl erstellt werden.

3.1.7.2.2 DSPPGM – Programm anzeigen

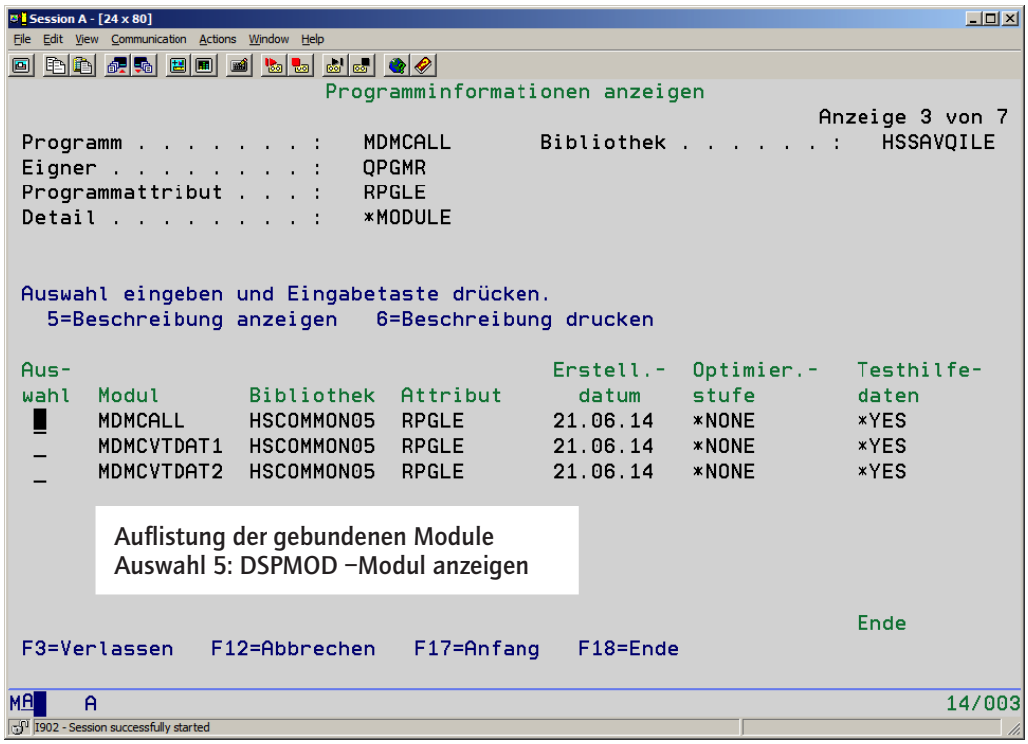
Mit dem CL-Befehl DSPPGM (Programm anzeigen) können Programminformationen angezeigt werden.

Besonders wichtig sind dabei die Informationen darüber, welche Module und welche Serviceprogramme gebunden sind. Diese Informationen können über die Detailauswahlen *MODULE (Auflistung der gebundenen Module) und *SRVPGM (Auflistung der gebundenen Serviceprogramme) ausgewählt und angezeigt werden.



DSPPGM – Programm anzeigen – Auswahl Detailinformationen

Das folgende Beispiel zeigt die gebundenen Module im Programm MDMCALL bei Auswahl der Detailinformation *MODULE. Neben anderen Informationen wird an dieser Stelle auch das Erstellungsdatum des Moduls angezeigt. Über Auswahl 5 können detaillierte Informationen über das Modul durch Ausführung des CL-Befehls DSPMOD (Modul anzeigen) angezeigt werden.

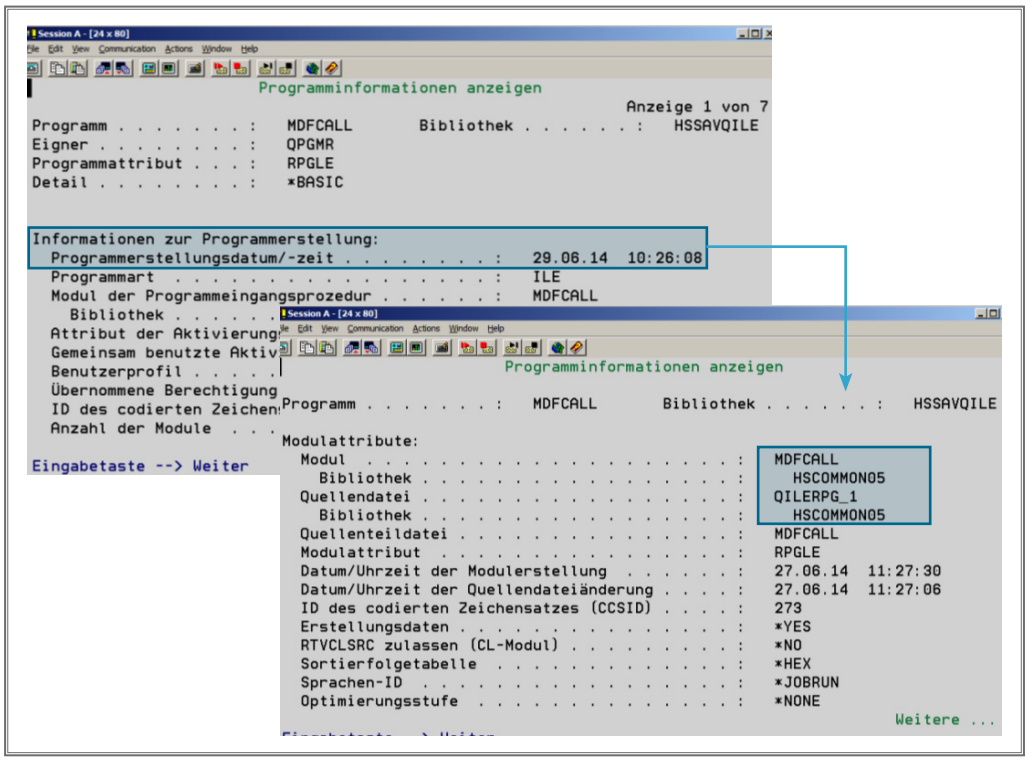


*DSPPGM – Option *MODULE = Modul-Informationen*

Quellen-Informationen, das heißt Informationen darüber, aus welcher Quelle das „Programm“ generiert wurde, können grundsätzlich nur über das Modul ermittelt werden. Unabhängig davon, ob ein Programm mit dem CL-Befehl CRTPGM (Programm erstellen) gebunden oder als Binder-Programm über einen CRTBNDxxx-Befehl erstellt wurde, wird immer eine zweistufige Kompilierung durchgeführt. Zunächst wird das Modul erstellt, und im Anschluss werden die Module in ein Programm gebunden.

3.1.7
Seite 20

Das folgende Beispiel zeigt einen Auszug aus dem Befehl DSPPGM. Das Modul, in der sich die Programm-Eingangs-Prozedur (PEP) befindet, wird unter den *BASIC-Informationen angezeigt. Nähere Informationen – zum Beispiel zu den Modulattributen – können jedoch nur aus dem Modul ermittelt werden und über Auswahl 5 (= Beschreibung anzeigen) angezeigt werden. Bei Auswahl 5 wird der DSPMOD-Befehl (Modul anzeigen) ausgeführt.



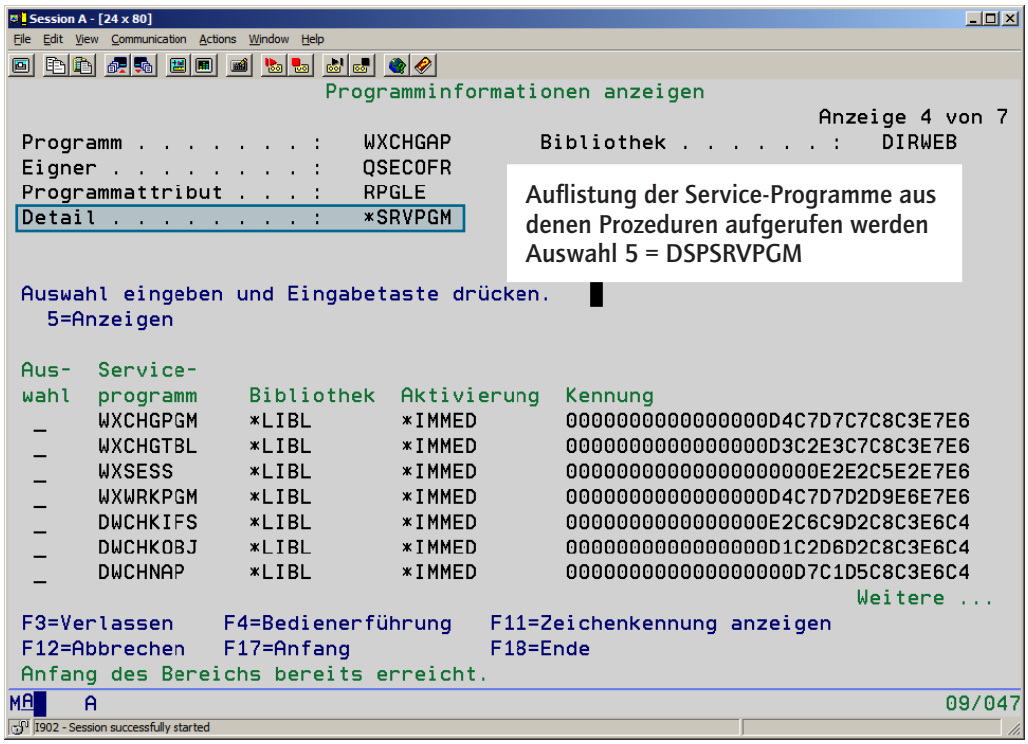
DSPPGM – Anzeigen von Quelleninformationen auf Modulebene

Fast ebenso wichtig, wie die Informationen über die gebundenen Module zu erhalten, ist zu wissen, aus welchen Serviceprogrammen Prozeduren aufgerufen werden, das heißt, welche Serviceprogramme gebunden wurden. Diese Informationen können über die Detailinformation *SRVPGM angezeigt werden.

In der folgenden Abbildung werden alle Serviceprogramme aufgelistet, aus denen im Programm WXCHGAP Prozeduren aufgerufen werden. Neben anderen Informationen wird an dieser Stelle auch angezeigt, ob das Serviceprogramm direkt bei Aufruf des rufenden Programms (Aktivierung = *IMMED) oder erst bei Aufruf der ersten Prozedur aus dem Serviceprogramm (Aktivierung = *DEFERRED) aktiviert wird.

Des Weiteren ist die eingebundene Signatur der aufgelisteten Serviceprogramme zu sehen.

Über Auswahl 5 (= Anzeigen) können über den Befehl DSPSRVPGM (Serviceprogramm anzeigen) detaillierte Informationen über das ausgewählte Serviceprogramm angezeigt werden.



DSPPGM – Auflistung der gebundenen Serviceprogramme

3.1.7.3 Verwendung von Modulen = modulare Programmierung?

Der Begriff modulare Programmierung wird gerne und häufig verwendet.

Doch was bedeutet modulare Programmierung tatsächlich? Die Verwendung von Modulen?

Die Antwort lautet NEIN.

Modulare Programmierung bedeutet, dass Programmteile, einzelne Zeilen, komplexe Konstrukte mit mehrfach verschachtelten Built-In-Funktionen, Subroutinen, Prozeduren, die mehrfach verwendet werden können, nur einmalig codiert werden und überall dort, wo sie benötigt werden, eingebunden bzw. aufgerufen werden können.

In der guten alten OPM-Zeit hatte man nur drei Möglichkeiten, um mit wiederverwendbarem Code umzugehen:

- Wiederverwendbarer Code wurde dupliziert, das heißt, die gleiche Folge von Befehlen wurde in jedem Programm, in dem sie benötigt wurde, auscodiert. Bei einer Änderung mussten alle Vorkommen gefunden und modifiziert werden, was zum einen ein zeitaufwendiger Prozess war. Zum anderen blieb auch (fast) immer das eine oder andere Vorkommen auf der Strecke.
- Der wiederverwendbare Source-Code wurde in ein Programm gepackt, das dynamisch aufgerufen wurde, Leider bringt der dynamische Programmaufruf einen gewissen Overhead mit sich, das heißt, verglichen mit dem Aufruf von Subroutinen sind Programmaufrufe zeitaufwendig.

  Des Weiteren musste pro Programm eine eigene Teildatei verwendet werden, die den Source Code enthielt. Programm- und Teildateinamen waren auf 10 Stellen beschränkt, was oft zu kryptischen Namen, einer Kombination von Buchstaben und Ziffern führte.

  Bedingt durch diese Restriktionen hielt sich die Anzahl der Unterprogramme in Grenzen.

- Die wiederverwendbaren Source Codes wurden in eigene Teildateien ausgelagert. Diese Programmteile wurden mit den Compiler-Direktiven /COPY oder /INCLUDE an allen Stellen eingebunden, an denen sie benötigt wurden.

  Bei einer Änderung musste der Source Code somit nur an einer einzigen Stelle modifiziert werden.

  Alle Programme, in denen die Copy-Strecke eingebunden wurde, mussten im Anschluss neu kompiliert werden.

  Nachteil von Copy-Strecken ist, dass sie nicht in sich gekapselt sind, das heißt, aus einer Copy-Strecke heraus können alle (globalen) Programmvariablen verändert werden, was mit einigem Risiko behaftet ist.

Mit Hilfe der ILE-Konzepte können die Probleme und Risiken der OPM-Codierung weitgehend vermieden werden.

- In der ILE-Welt kann man bereits durch das direkte Binden von Modulen einige der zuvor genannten Probleme und Risiken der OPM-Methoden umgehen.

  Das Binden von Modulen ähnelt auf den ersten Blick einer Kombination aus Programmaufruf und Copy-Strecke. Der Source Code befindet sich wie bei einer Copy-Strecke in einer eigenen Teildatei. Anstatt jedoch diese Teildatei direkt in dem Programm-Quell-Code zu hinterlegen, wird aus der Teildatei ein Modul erstellt, das aus dem Programm-Code über einen CALL-Befehl aufgerufen wird. Im Gegensatz zum Aufruf eines Programms, das ein eigenständiges Objekt bleibt, wird das Modul im Binderschritt physisch in das Programm-Objekt kopiert.

  Der Source Code eines Modules ist in einer Teildatei gekapselt, das heißt, alle Variablen werden innerhalb des Moduls definiert und können auch nur innerhalb des Moduls verändert werden.

  Der Aufruf des Moduls bzw. der Main- oder exportierten Prozeduren erfolgt über einen CALL-Befehl, und alle Informationen werden nur über Parameter ausgetauscht. Der Overhead, der bei einem dynamischen Programmaufruf entsteht, entfällt jedoch, da das Modul durch das Binden (Bind by Copy) in das Programmobjekt kopiert und somit Bestandteil des Programms wird.

  Der Nachteil beim direkten Binden von Modulen in Programme ist, dass bei Änderung eines Moduls dieses Modul in allen Programmen, in denen es gebunden ist, ausgetauscht werden muss. Damit ist ein ähnlicher Aufwand wie bei der Aktualisierung von Copy-Strecken erforderlich, was diese Art der Bindung nicht besonders attraktiv macht.

- Anstatt die Module direkt in Programme zu binden, können sie auch in Serviceprogramme gebunden werden. Prozeduren, die aus Service-Programmen exportiert wurden, können aus jedem beliebigen Programm und jeder Prozedur, die entweder im gleichen oder einem anderen Service-Programm hinterelegt ist, aufgerufen werden.

  Beim Binden wird anstatt des Moduls lediglich die eindeutige Kennung (Symbol/Signatur) des Serviceprogramms eingebunden. Wird das Modul geändert, muss es lediglich in dem Serviceprogramm, in dem es gebunden ist, ausgetauscht werden. Da in den Programmen und Serviceprogrammen, aus denen Prozeduren aus dem geänderten Serviceprogramm aufgerufen werden, nur die Signatur des Serviceprogramms eingebunden ist, ist bei Änderung des Moduls weder ein Austausch noch ein erneutes Binden/Erstellen erforderlich.

  Beim Aufruf der ersten Prozedur aus einem Serviceprogramm oder beim Aktivieren des rufendes Programms wird das Serviceprogramm anhand der hinterlegten Signatur ermittelt und in den Speicher geladen. Wann die Aktivierung erfolgt, hängt von der verwendeten Aktivierungsgruppe ab. Das Serviceprogramm bleibt so lange im Speicher wie die Aktivierungsgruppe, in der es läuft, aktiv ist. Aus diesem Grund ist der Aufruf einer Prozedur aus einem Serviceprogramm annähernd so schnell wie der Aufruf einer internen Prozedur oder einer Subroutine.

  Das Ändern von Prozeduren in Serviceprogrammen ist somit kein Problem. Beim Hinzufügen neuer Prozeduren ändert sich jedoch per Default die eindeutige Signatur des Serviceprogramms. Die Folge sind Signaturverletzungen beim Aktivieren der Serviceprogramme.

  Diese Probleme kann man jedoch durch die Verwendung der Binder-Sprache umgehen. Mit Hilfe der Binder-Sprache kann die Signatur eines Serviceprogramms fest vorgegeben werden. Alternativ können auch mit Hilfe der Binder-Sprache zusätzlich zur aktuellen Signatur die vorherigen Signaturen im Serviceprogramm gespeichert werden. In beiden Fällen wird das Serviceprogramm bei der Aktivierung gefunden.

  Mit diesen Methoden steht einer hochmodularen Software-Entwicklung nichts mehr im Weg.

3.1.7.3.1 Von der Copy-Strecke zum Modul

Wiederverwendbarer Source Code wurde entweder immer wieder codiert oder in eine Copy-Strecke ausgelagert.

Unter einer Copy-Strecke versteht man ein Stück Source Code, das in einer eigenen Teildatei gespeichert ist. Dieser „wiederverwendbare" Source Code bzw. die Teildatei, in die er ausgelagert wurde, wird mit Hilfe einer der Compiler-Anweisungen /COPY oder /INCLUDE in andere Teildateien an jeder Stelle im Source Code eingebunden, an der er benötigt wird.

Der Nachteil von Copy-Strecken ist, dass sie in sich nicht gekapselt sind. In Copy-Strecken kann und muss auf Variablen zugegriffen werden, die im Source Code, in den sie eingebunden werden, definiert sind. Bei einer Änderung der Copy-Strecke muss man sich über die Auswirkung auf alle Programme, in der die Copy-Strecke eingebunden ist, im Klaren sein.

Verwendet man statt einer Copy-Strecke ein Modul, wird das Modul zwar auch in das Programm kopiert, das Modul ist jedoch in sich gekapselt und Informationen zwischen Programm- und Modul-Source-Code werden nur über Parameter ausgetauscht.

3.1.7.3.1.1 Ursprünglicher Source Code mit Copy-Strecken

In einer Anwendung wurden zwei Copy-Strecken definiert:

- CPYCVTDAT1:
  Datum drehen von Format JJJJMMTT in TTMMJJJJ und umgekehrt
- CPYCVTDAT2:
  Aufbereiten eines numerischen Datums im Format JJJJMMTT

Beide Copy-Strecken wurden im Programm CPYCALL eingebunden.

Der folgende Source Code zeigt den Inhalt von Teildatei CPYCVTDAT1 (Datum drehen).

Die verwendeten Variablen DatumTMJJ und DatumJJMT werden beide im Source Code definiert. (1)

Sofern die Variable DatumTMJJ einen positiven numerischen Wert enthält, wird davon ausgegangen, dass es sich bei dem Inhalt um ein numerisches Datum im Format TTMMJJJJ handelt. Dieses numerische Datum wird gedreht, das heißt in ein numerisches Datum im Format JJJJMMTT konvertiert. (2)

3.1.7
Seite 26

Sofern die Variable DatumJJMT einen positiven numerischen Wert enthält und die Variable DatumTMJJ keinen positiven Wert enthalten hatte, wird davon ausgegangen, dass die Variable DatumJJMT ein numerisches Datum im Format JJJJMMTT enthält. Dieses numerische Datum wird gedreht, das heißt in ein numerisches Datum im Format TTMMJJJJ konvertiert. (3)

Bei einem ungültigen numerischen Datum werden beide Variablen initialisiert, das heißt auf *ZEROS gesetzt. (4)

```
C           Monitor
C           Select
C           When    DatumTMJJ > *Zeros                    (2)
C           Eval    DatumJJMT = %Dec(%Date(DatumTMJJ: *EUR):
C                                    *ISO)
C           When    DatumJJMT > *Zeros                    (3)
C           Eval    DatumTMJJ = %Dec(%Date(DatumJJMT: *ISO):
C                                    *EUR)
C           EndSL
C           On-Error
C           Clear             DatumTMJJ       8 0         (1)
C           Clear             DatumJJMT       8 0         (4)
C           EndMon
```

Copy-Strecke – Teildatei CPYCVTDAT1 – Datum drehen

Der folgende Source Code zeigt die Copy-Strecke CPYCVTDAT2 (Datum aufbereiten).

Innerhalb des Source Codes wurde keine Variable definiert, das heißt, alle verwendeten Variablen müssen mit dem gewünschten Namen und der erwarteten Definition im Source Code der rufenden Programme hinterlegt werden.

Das numerische Datum in Variable DatumJJMT wird zunächst in ein echtes Datum (Datum) konvertiert. (1)

Abhängig vom Format in Variable DatFmtA wird das konvertierte Datum (DATUM) in eine alphanumerische Darstellung eines Datums (DatumA) im vorgegebenen Format (DatFmtA) konvertiert. (2)

Im Anschluss werden die in Variable DATSEP hinterlegten Trennzeichen in das alphanumerische Datum integriert. (3)

```
C           Monitor
C           Clear                   DatumA
C           Clear                   Datum
C
C           Eval    Datum = %Date(DatumJJMT: *ISO)        (1)
C
C           Select                              (2)
C           When    DatFmtA = 'JJMT'
C           EvaL    DatumA  = %Char(Datum: *ISO)
C           When    DatFmtA = 'TMJJ'
C           EvaL    DatumA  = %Char(Datum: *EUR)
C           When    DatFmtA = 'MTJJ'
C           EvaL    DatumA  = %Char(Datum: *USA)
C           When    DatFmtA = 'JMT'
C           EvaL    DatumA  = %Char(Datum: *YMD)
C           When    DatFmtA = 'TMJ'
C           EvaL    DatumA  = %Char(Datum: *DMY)
C           When    DatFmtA = 'MTJ'
C           EvaL    DatumA  = %Char(Datum: *MDY)
C           EndSl
C
C           If      DatFmtA <> 'JJMT'                     (3)
C           Eval    DatumA = %Subst(DatumA: 1: 2) + DatSep +
C                            %Subst(DatumA: 4: 2) + DatSep +
C                            %Subst(DatumA: 7)
C           Else
C           Eval    DatumA = %Subst(DatumA: 1: 4) + DatSep +
C                            %Subst(DatumA: 6: 2) + DatSep +
C                            %Subst(DatumA: 9)
C           EndIf
C
C           On-Error
C           Clear                   DatumA
C           EndMon
```

Copy-Strecke CPYCVTDAT2 – Datum aufbereiten

3.1.7
Seite 28

Beide Copy-Strecken wurden im Programm CPYCALL eingebunden, dessen Source Code im folgenden Beispiel gezeigt wird.

Die Variablen, die in den Copy-Strecken CPYCVTDAT1 und CPYCVTDAT2 zwar verwendet, jedoch in diesen nicht definiert werden, werden in Teildatei CPYCALL definiert. (1)

Die Variable DatumTMJJ wird mit dem numerischen Tagesdatum im Format TTMMJJJJ initialisiert. (2) Die Variable DatumTMJJ wurde in Teildatei CPYCALL nicht definiert. Da sie jedoch in Copy-Strecke CPYCVTDAT1 definiert ist, kann das Programm ordnungsgemäß kompiliert werden.

Danach wird die Copy-Strecke CPYCVTDAT1 mit Hilfe der Compiler-Direktive /COPY integriert. (3) Vor Release 7.1 TR 7 ist die Angabe der /End-Free und /Free Compiler-Direktiven erforderlich, da der Programmcode im Free-Format codiert ist, die Copy-Strecken jedoch im spaltenorientierten Format (4).

Das gedrehte Datum in Variable DATUMJJMT wird im Anschluss angezeigt.

Die Variablen DatFmtA und DatSep werden mit dem Format (JJMT ➜ JJJJ-MM-TT) und dem Trennzeichen (-) für die alphanumerische Aufbereitung initialisiert (5).

Zur alphanumerischen Aufbereitung des Datums wird der Inhalt der Copy-Strecke CPYCVTDAT2, die über die Compiler-Direktive /COPY integriert wurde, herangezogen. (6)

Nach der Anzeige des aufbereiteten Datums erfolgt eine zweite Aufbereitung und Anzeige im Format TTMJ mit Trennzeichen / ➜ TT/MM/JJJJ. Die Copy-Strecke CPYCVTDAT2 wird an dieser Stelle ein zweites Mal eingebunden. (7)

```
D Datum           S               D                          (1)
D DatumA          S              10A
D DatumJJMT       S               8P 0
D DatFmtA         S               4A
D DatSep          S               1A

D PGMSDS         SDS                     Qualified
D    MsgTxt              91    140A
 *---------------------------------------------------------------
 /Free
   *INLR = *ON;

   Clear DatumJJMT;
   DatumTMJJ = %Dec(%Date(): *EUR);                             (2)

 /End-Free                                                      (4)
 /Copy QILESRC,CpyCvtDat1                                       (3)
 /Free                                                          (4)

   Dsply DatumJJMT;
   DatFmtA= 'TMJJ';
   DatSep = '-';                                                (5)

 /End-Free
 /Copy QILESRC,CpyCvtDat2                                       (6)
 /Free

   Dsply DatumA;
   Clear DatumA;
   DatFmtA= 'JJMT';
   DatSep = '/';

 /End-Free
 /Copy QILESRC,CpyCvtDat2                                       (7)
 /Free

   Dsply DatumA;
   Return;
 /End-Free
```

Programm CPYCALL mit integrierten Copy-Strecken

Das Programm CPYCALL kann direkt mit dem Befehl CRTBNDRPG (RPG-Binder-Programm erstellen) erstellt werden, wie das folgende Beispiel zeigt:

```
CRTBNDRPG PGM(YOURLIB/CPYCALL)
          SRCFILE(QILESRC_I)
```

Erstellen des Programms CPYCALL mit integrierten Copy-Strecken

3.1.7.3.1.2 Modifizierter Source Code: Modul statt Copy-Strecke

Der Nachteil bei der Verwendung von Copy-Strecken ist, dass die Variablen mit dem erwarteten Namen und Format in der rufenden Prozedur/Programm definiert werden müssen und aus der Copy-Strecke globale Variable verändert werden können. Des Weiteren werden nur die Variablennamen und die Kompatibilität des Datentyps geprüft, nicht jedoch die Länge. Abweichende Variablendefinitionen können zu unerwarteten bzw. falschen Ergebnissen führen.

Konvertiert man dagegen die Copy-Strecken in Main-Procedures, kompiliert diese modifizierten Teildateien in Module und bindet diese Module in das Programm, so können Variablen nur innerhalb des Moduls geändert werden. Die Benennung der Variablen im rufenden Programm und innerhalb des Moduls muss nicht übereinstimmen, da Austausch von Informationen beim Aufruf der Main-Procedure nur über Parameter erfolgt.

Sofern Prototyping zur Definition und Übergabe der Parameter verwendet wird, kann eine Parameterprüfung zur Compile-Zeit erfolgen, das heißt, bei inkorrekter Parameterdefinition oder -übergabe kann das Programm nicht erstellt werden. Auch alle anderen Vorteile von Prototyping – wie die Übergabe von Input-Only-Parametern oder die Verwendung von optionalen Parametern – können genutzt werden.

Der folgende Source Code zeigt die modifizierte Quelle der Copy-Strecke CPYCVTDAT1

Da die Informationen über Parameter übergeben werden, wird ein Procedure Interface mit den Ein/Ausgabe-Parametern DatumTMJJ und DatumJJMT integriert. (1)

Um eine Parameterprüfung zur Compile-Zeit zu ermöglichen, wird zusätzlich ein Prototyp erstellt und in der Copy-Strecke MDMProto hinterlegt. Diese Copy-Strecke muss in alle RPGLE-Quellen eingebunden werden, aus denen die Main-Procedure MDMCVTDAT1 aufgerufen wird. (2)

Da Programme und/oder Module mit Main-Procedures nur dann umgewandelt werden können, wenn ein Beendigungscode (*INLR oder RETURN) gefunden wird, wird am Ende der Prozedur der Operations-Code RETURN ausgeführt. (3)

```
D/Copy QILESRC,MDMProto                                          (2)

 * Procdure Interface
D MDMCVTDAT1   PI                                                (1)
D   DatumTMJJ           8P 0
D   DatumJJMT           8P 0
 *---------------------------------------------------------------
C            Monitor

C            Select
C            When    DatumTMJJ > *Zeros
C            Eval    DatumJJMT = %Dec(%Date(DatumTMJJ: *EUR):
C                                     *ISO)
C            When    DatumJJMT > *Zeros
C            Eval    DatumTMJJ = %Dec(%Date(DatumJJMT: *ISO):
C                                     *EUR)
C            EndSL
C            On-Error
C            Clear                DatumTMJJ
C            Clear                DatumJJMT
C            EndMon
C            Return                                              (3)
```

Modul MDMCVTDAT1 (Datum drehen) mit Main-Procedure

Der in Teildatei MDMProto hinterlegte Prototyp entspricht dem Procedure Interface – mit einer kleinen Abweichung. Anstatt einer PI-Zeile wurde eine PR-Zeile verwendet, wie der folgende Source Code zeigt:

```
D MDMCVTDAT1      PR
D   DatumTMJJ           8P 0
D   DatumMMMT           8P 0
```

Prototyp für die Main-Procedure in Modul MDMCVTDAT1

Ergänzung 32/2015

3.1.7
Seite 32

Die Teildatei kann im Anschluss mit Hilfe des Befehl CRTRPGMOD (Modul erstellen) in ein Modul umgewandelt werden. Das Modul MDMCVTDAT1 kann mit dem folgenden Befehl generiert werden:

```
CRTRPGMOD MODULE(YOURLIB/MDMCVTDAT1)
          SRCFILE(YOURSRCLIB/YOURSRCFIL)
          SRCMBR(MDMCVTDAT1)
```

Erstellen des Moduls MDMCVTDAT1

Das folgende Beispiel zeigt den modifizierten Source Code von Copy-Strecke CPYCVTDAT2. Der modifizierte Source Code wurde in Teildatei MDMCVTDAT2 hinterlegt.

Damit die Informationen mit dem rufenden Programm über Parameter ausgetauscht werden können, wurden die Ein-/Ausgabe-Parameter in einem Procedure Interface hinterlegt (1).

Der Prototyp für MDMCVTDAT2 wurde wie beim vorangegangenen Modul in Teildatei MDMProto hinterlegt (2).

Die Namen der Ein-/Ausgabe-Parameter bzw. der intern verwendeten Variablen wurden gegenüber der ursprünglichen Copy-Strecke verändert; beispielsweise wird statt DatumJJMT ParDateYYMD verwendet (3).

Des Weiteren wurden die ersten drei Parameter durch das Schlüsselwort CONST als Input-Only-Parameter definiert (4). Damit können in der rufenden Prozedur Variablen mit kompatiblen Datentypen, jedoch abweichender Länge, Konstanten und sogar Ausdrücke übergeben werden.

Die Main-Procedure wird, wie auch beim vorangegangenen Modul mit RETURN beendet (5).

Da die Main-Procedure mit RETURN beendet wird, werden beim erneuten Aufruf die in den globalen D-Bestimmungen definierten Variablen nicht erneut initialisiert. Um zu vermeiden, dass die Variable DATUM falsche/alte Werte enthält, wird sie direkt zu Beginn der Prozedur initialisiert, das heißt durch den CLEAR-Befehl auf den 01.01.0001 gesetzt (6).

```
D/INCLUDE QILESRC,MDMProto                                    (2)
 * Procdure Interface
D MDMCVTDAT2    PI                                            (1)
D   ParDateYYMD               8P 0 Const                     (3)
D   ParDatFmtA                4A   Const                     (4)
D   ParDatSep                 1A   Const
D   POutDatumA               10A

D Datum         S                D
 *---------------------------------------------------------------
C              Monitor
C              Clear                 POutDatumA
C              Clear                 Datum                   (6)
C
C              Eval      Datum = %Date(ParDateYYMD: *ISO)
C
C              Select
C              When      ParDatFmtA = 'JJMT'
C              EvaL      POutDatumA  = %Char(Datum: *ISO)
C              When      ParDatFmtA = 'TMJJ'
C              EvaL      POutDatumA  = %Char(Datum: *EUR)
C              When      ParDatFmtA = 'MTJJ'
C              EvaL      POutDatumA  = %Char(Datum: *USA)
C              When      ParDatFmtA = 'JMT'
C              EvaL      POutDatumA  = %Char(Datum: *YMD)
C              When      ParDatFmtA = 'TMJ'
C              EvaL      POutDatumA  = %Char(Datum: *DMY)
C              When      ParDatFmtA = 'MTJ'
C              EvaL      POutDatumA  = %Char(Datum: *MDY)
C              EndSl
C
C              If        ParDatFmtA <> 'JJMT'
C              Eval      POutDatumA =
C                             %Subst(POutDatumA: 1: 2) +
                                      ParDatSep +
C                             %Subst(POutDatumA: 4: 2) +
                                      ParDatSep +
C                             %Subst(POutDatumA: 7)
```

3.1.7
Seite 34

```
C              Else
C              Eval      POutDatumA =
C                                %Subst(POutDatumA: 1: 4) +
                                        ParDatSep +
C                                %Subst(POutDatumA: 6: 2) +
                                        ParDatSep +
C                                %Subst(POutDatumA: 9)
C              EndIf
C
C              On-Error
C              Clear                 POutDatumA
C              EndMon

C              Return                                    (5)
```

Modul MDMCVTDAT2 (Datum aufbereiten) mit Main-Procedure

Das folgende Code-Beispiel zeigt den Prototyp für die Main-Procedure in Teildatei MDMCVTDAT2, der in Teildatei MDMProto hinterlegt wurde.

Da nur die Anzahl, Datentypen und Schlüsselworte geprüft werden, ist die Angabe von Parameternamen optional, das heißt, es können entweder keine oder beliebige Parameternamen verwendet werden.

Bei der Definition des Prototyps für die Main-Procedure in MDMCVTDAT2 wurden für die Input-Parameter keine Paramternamen (1) angegeben, während für den Ein-/Ausgabe-Parameter ein vom Procedure Interface abweichender Name (2) verwendet wurde.

```
D MDMCVTDAT2      PR
D                              8P 0 Const
D                              4A   Const                  (1)
D                              1A   Const
D   ParOut01                  10A                          (2)
```

Prototyp für Main-Procedure in Teildatei MDMCVTDAT1

Das Modul kann im Anschluss mit dem folgenden CL-Befehl erstellt werden:

```
CRTRPGMOD MODULE(YOURLIB/MDMCVTDAT2)
          SRCFILE(YOURSRCLIB/YOURSRCFIL)
          SRCMBR(MDMCVTDAT2)
```

Erstellen Modul MDMCVTDAT1

Das folgende Beispiel zeigt das Programm MDMCALL, in dem die in den

Modulen MDMCVTDAT1 und MDMCVTDAT2 hinterlegten Main-Procedures aufgerufen werden.

Die Prototypen für die beiden Main-Procedures sind in der Teildatei MDMProto hinterlegt. Da die Main-Procedures in diesem Programm aufgerufen werden sollen, müssen zum Abgleich der Parameter die Prototypen eingebunden werden (1).

Nach den Prototypen werden die Variablen definiert, die im Programm verwendet werden (2). Da die Module in sich gekapselt sind und der Informationsaustausch nur über Parameter erfolgt, muss an dieser Stelle keine Rücksicht auf die Namensgebung innerhalb der Module genommen werden.

Wie im ursprünglichen Source Code wird die Variable DatumDMYY mit dem numerischen Tagesdatum im Format TTMMJJJJ initialisiert. (3)

Im Anschluss daran wird die Main-Procedure in Modul MDMCVTDAT1 mit Hilfe des Operations-Codes CALLP aufgerufen, dessen Angabe im RPG-Free-Format optional ist. Das zu konvertierende numerische Datum in Variable DatumDMYY wird als erster Parameter definiert. Die Variable DatumJJMT in der das konvertierte Datum zurückgegeben wird, wird als zweiter Parameter übergeben. Das konvertierte Datum wird anschließend angezeigt (4).

Vor dem Aufruf der Main-Procedure in MDMCVTDAT2 werden die Variablen DatFmtA und DatSep mit dem gewünschten Format (JJMT) und dem gewünschten Separator (/) initialisiert. (5) Die Main-Procedure in Modul MDMCVTDAT2 wird ohne Angabe des Operations-Codes CALLP, der im Free-Format optional ist, aufgerufen. (6) Das aufbereitete Datum wird im Anschluss angezeigt.

Da die ersten drei Parameter in der Main-Procedure von MDMCVTDAT2 über das Schlüsselwort CONST als Input-Only-Parameter definiert wurden, können wie beim zweiten Aufruf der Main-Procedure MDMCVTDAT2 die Parameterwerte direkt übergeben werden (7).

3.1.7
Seite 36

```
D/INCLUDE QILESRC,MDMProto                                    (1)

D DatumA          S              10A                          (2)
D DatumJJMT       S               8P 0
D DatumDMYY       S               8P 0
D DatFmtA         S               4A
D DatSep          S               1A

D PGMSDS         SDS                     Qualified
D    MsgTxt               91    140A
 *---------------------------------------------------------------
 /Free
   *INLR = *ON;

   Clear DatumJJMT;
   DatumDMYY = %Dec(%Date(): *EUR);                           (3)
   MDMCvtDat1(DatumDMYY: DatumJJMT);                          (4)
   Dsply DatumJJMT;

   DatFmtA= 'TMJJ';                                           (5)
   DatSep = '-';
   MDMCvtDat2(DatumJJMT: DatFmtA: DatSep: DatumA);            (6)
   Dsply DatumA;

   Clear DatumA;
   MDMCvtDat2(20140701: 'JJMT': '/': DatumA);                 (7)
   Dsply DatumA;

   Return;
/End-Free
```

MDMCALL – Aufruf von Main-Prozeduren in Modulen

Um das Programm erstellen zu können, müssen zunächst alle drei Module über den Befehl CRTMOD (Modul erstellen) generiert werden.

Die umgewandelten Module müssen anschließend mit dem Befehl CRTPGM (Programm erstellen) zu einem Programm gebunden werden. Dabei müssen die einzelnen Module aufgelistet und das Eingangsmodul festgelegt werden.

Das Programm MDMCALL kann mit dem folgenden Befehl aus den vorhandenen Modulen erstellt werden:

```
CRTPGM PGM(YOURJLIB/MDMCALL)
       MODULE(MDMCALL MDMCVTDAT1 MDMCVTDAT2)
       ENTMOD(*FIRST)
```

Binden von Programm MDMCALL aus mehreren Modulen

Man kann sich vorstellen, dass bei dieser Technik in Verbindung mit einer sehr modularen Programmierung zu jedem Programm eine große Anzahl Module gebunden werden muss. Sofern die Module beim Binden einzeln auflistet werden müssen, kann man sehr schnell den Überblick verlieren, und mal das eine oder andere Modul vergessen, wodurch das Programm nicht erstellt werden kann.

Diesem Problem kann man auf drei Arten vorbeugen:

- Für jedes Programm wird ein CL-Programm erstellt, in dem alle notwendigen Schritte ausgeführt werden. Ändert sich ein Modul, müssen alle CL-Programme, in denen das Modul aufgelistet ist, ausgeführt werden. Das/die eigentlichen Programme werden dann durch Ausführung der CL-Programme neu erstellt.
- Anstatt Module mit nur einer einzigen Main-Procedure zu verwenden, setzt man exportierte Prozeduren ein. In einer einzigen Teildatei können auf diese Weise mehrere unabhängige interne und exportierte Prozeduren hinterlegt werden. Nur die exportierten Prozeduren können von außerhalb des Moduls aufgerufen werden.

  Damit kann man mehrere Prozeduren abhängig von ihrer Funktionalität in einer einzigen Teildatei gruppieren (zum Beispiel Datumsfunktionen, String-Funktionen, Zugriffe auf eine bestimmte Datei/Tabelle etc.) und so die Anzahl der Module beträchtlich reduzieren.

  Dennoch wird eine nicht unerhebliche Anzahl Module übrig bleiben, die individuell in unterschiedliche Programme gebunden werden müssen.

3.1.7
Seite 38

- Um zum einen zu vermeiden, dass für jedes Programm ein zusätzliches CL-Programm erstellt werden muss und um zum anderen Probleme zu umgehen, die bei der Auflistung von Modulen entstehen können, kann ein Binder-Verzeichnis verwendet werden. In das Binder-Verzeichnis werden alle Module eingetragen.

  Beim Erstellen eines Programms kann kein, ein einziges oder mehrere Binder-Verzeichnisse angegeben werden. Im Binder-Schritt werden die Einträge in den zugeordneten Binder-Verzeichnissen bzw. die zugehörigen Objekte (Module oder Serviceprogramme) nach den im Programm aufgerufenen Prozeduren durchsucht. Wird die Prozedur gefunden, so wird das entsprechende Modul (by Copy) oder die Signatur des Serviceprogramms (Bind by Reference) in das Programm kopiert.

3.1.7.3.2 Von der Main-Procedure zur Sub-Procedure

In den vorangegangenen Beispielen wurden die ursprünglichen Copy-Strecken in Main-Procedures umgewandelt, die jeweils in eigenen Teildateien hinterlegt sind.

Da für jede Prozedur eine eigene Teildatei verwendet werden müsste, und die Prozedurnamen weiterhin auf 10 Zeichen beschränkt sind, eignet sich diese Vorgehensweise nur bedingt für eine modulare Programmierung.

Verwendet man anstatt Main-Procedures echte exportierte Prozeduren, so können mehrere Prozeduren mit ähnlicher Funktionalität zusammengruppiert werden, zum Beispiel alle Datumsfunktionen in einer Teildatei/Modul.

Um die Anzahl der Module zu reduzieren, werden im folgenden Beispiel die beiden ursprünglichen Main-Procedures in exportierte Prozeduren verwandelt und in der gleichen Teildatei (MDSCVTDAT) hinterlegt.

Da in der Quelle keine Main-Procedure (globale C-Bestimmungen) mehr vorhanden ist, sondern direkt mit einer P-Bestimmung, also der ersten Prozedur begonnen wird, muss das Schlüsselwort „NoMain“ in den H-Bestimmungen angegeben werden (1). Ohne die Angabe des Schlüsselworts NoMain versucht der Compiler den RPG-Zyklus zu integrieren. Da keine Main-Procedure vorhanden ist, kann der Compiler auch nicht feststellen, wie das Programm beendet wird, und die Kompilierung wird fehlerhaft beendet.

Die Prototypen für die beiden exportierten Prozeduren wurden in der Teildatei MDSProto hinterlegt, die über die Compiler-Anweisung /COPY in die Quelle eingebunden wurde.

(Sub-)Procedures beginnen und enden im fixen Format immer mit einer P-Bestimmung (3). Im Free-Format wird der Source Code der Prozedur zwischen ein DCL-PROC- und ein END-PROC-Statement eingebettet (4). Die wiederholte Angabe des Prozedurnamens ist bei den fixen P-Bestimmungen erforderlich, beim Operations-Code END-PROC jedoch optional.

Da (Sub-)Prozedurnamen nicht auf 10 Zeichen beschränkt sind, wurden für dieses Beispiel längere und sprechendere Namen gewählt. MDS_DatumDrehen entspricht der Main-Procedure/Modul MDMCVTDAT1, während MDS_DatumAufbereiten der Main-Procedure/Modul MDMCVTDAT2 entspricht.

Bei der Prozedur MDS_DrehenDatum wird bei den P- und D-Bestimmungen noch die Fix-Format-Codierung verwendet und nur die C-Bestimmungen werden im Free-Format codiert. Die Prozedur MDS_AufbereitenDatum dagegen ist komplett im Free-Format geschrieben.

Damit eine Prozedur von außerhalb des Moduls aufgerufen werden kann, ist die Angabe des Schlüsselworts EXPORT (5) beim beginnenden P-Statement bzw. beim Operations-Code DCL-PROC erforderlich.

Die Prototypen für beide Prozeduren sind in der Teildatei MDSProto hinterlegt.

Während an die Main-Procedure MDMCVTDAT1 nur zwei Parameter übergeben wurden, wurden für die Prozedur MDS_DatumDrehen drei Parameter definiert.

Sowohl der zweite (PInNumDate = Eingangsdatum) als auch der dritte Parameter (PInFmt = Format) wurden durch das Schlüsselwort CONST als Input-Only-Parameter definiert. Diese Parameter werden innerhalb des Source Codes der Prozedur nicht verändert. Eine Änderung der Parametervariablen innerhalb des Source Codes würde zu einem Kompilierungsfehler führen.

Der dritte Parameter (PInFmt = Format) wurde außerdem als optionaler Parameter definiert. Innerhalb des Source Codes wird mit Hilfe der Built-In-Funktionen %PARMS (Anzahl übergebene Parameter) und %PARMNUM (Position des Parameters) geprüft, ob dieser Parameter übergeben wurde oder nicht (6). Wurde er übergeben, so wird das übergebene Format verwendet, anderenfalls wird das Default-Format JJMT verwendet.

Durch die Monitor-Group (7) wird jeder Fehler abgefangen, der innerhalb der Prozedur auftreten könnte. Tritt ein Fehler auf, so wird das numerische Datum 99991231 (31.12.9999) ausgegeben.

Der Source Code für die Funktion MDS_DatumAufbereiten entspricht weitgehend dem Source Code der Main-Procedure in MDMCVTDAT2.

Da die Main-Procedure MDMCVTDAT2 mit RETURN verlassen wurde, werden beim erneuten Aufruf die Variablen, die in den globalen D-Bestimmungen definiert sind, nicht automatisch initialisiert. Aus diesem Grund wurde die Variable DATUM zu Beginn der Main-Procedure initialisiert.

Beim Aufruf einer (Sub-)Procedure werden die lokalen Variablen per Default immer initialisiert. Sofern der Variablenwert über mehrere Aufrufe konstant gehalten, also nicht initialisiert werden soll, muss die Variable explizit mit dem Schlüsselwort STATIC definiert sein. Da in Prozedur MDS_DatumAufbereiten die lokalen Variablen per Default initialisiert werden, wurde die Variable nicht (mehr) in den C-Bestimmungen initialisiert.

```
H NoMain                                                        (1)
 ****************************************************************
 /Copy QILESRC,MDSProto                                         (2)
 ****************************************************************
 * Procedure name: MDS_DatumDrehen
 ****************************************************************
P MDS_DatumDrehen...                                            (3)
P                 B                   Export                    (5)

D MDS_DatumDrehen...
D                 PI
D  POutNumDate                 8P 0
D  PInNumDate                  8P 0 CONST
D  PInFmt                      4A   CONST OPTIONS(*NOPASS)

D ParFmt          S            4A   inz('JJMT')
D*---------------------------------------------------------------
 /FREE
   Monitor;                                                     (7)
     If PInNumDate = *Zeros;
        Return;
     EndIf;

     If %Parms >= %ParmNum(PInFmt)
        and PInFmt <> *Blanks;                                  (6)
        ParFmt = PInFmt;
     EndIf;

     If ParFmt = 'TMJJ';
        POutNumDate = %Dec(%Date(PInNumDate: *EUR): *ISO);
     Else;
        POutNumDate = %Dec(%Date(PInNumDate: *ISO): *EUR);
     EndIf;

   On-Error;                                                    (7)
     POutNumDate = 99991231;
   EndMon;                                                      (7)
```

3.1.7
Seite 42

```
    Return;
  /End-Free
P MDS_DatumDrehen...
P                 E                                          (3)
  //*************************************************************
  // Procedure name: MDS_DatumAufbereiten
  //*************************************************************
  Dcl-Proc MDS_DatumAufbereiten Export;                (4)(5)

  DCL-PI *N;
      ParDateYYMD Packed(8)   Const;
      ParDatFmtA  Char(4)     Const;
      ParDatSep   Char(1)     Const;
      POutDatumA  Char(10);
  End-PI;

  DCL-S Datum Date;
  //-------------------------------------------------------------
    Monitor;
      Clear POutDatumA;

      Datum = %Date(ParDateYYMD: *ISO);

      Select;
      When ParDatFmtA = 'JJMT';
           POutDatumA  = %Char(Datum: *ISO);
      When ParDatFmtA = 'TMJJ';
           POutDatumA  = %Char(Datum: *EUR);
      When ParDatFmtA = 'MTJJ';
           POutDatumA  = %Char(Datum: *USA);
      When ParDatFmtA = 'JMT';
           POutDatumA  = %Char(Datum: *YMD);
      When ParDatFmtA = 'TMJ';
           POutDatumA  = %Char(Datum: *DMY);
      When ParDatFmtA = 'MTJ';
           POutDatumA  = %Char(Datum: *MDY);
      EndSl;
```

```
      If ParDatFmtA <> 'JJMT';
        POutDatumA = %Subst(POutDatumA: 1: 2) + ParDatSep +
                     %Subst(POutDatumA: 4: 2) + ParDatSep +
                     %Subst(POutDatumA: 7);
      Else;
        POutDatumA = %Subst(POutDatumA: 1: 4) + ParDatSep +
                     %Subst(POutDatumA: 6: 2) + ParDatSep +
                     %Subst(POutDatumA: 9);
      EndIf;
    On-Error;
      Clear POutDatumA;
    EndMon;

    Return;
  End-Proc MDS_DatumAufbereiten;                          (4)
```

Individuelle Prozeduren in einem Modul

Der folgende Source Code zeigt die beiden Prototypen, die in Teildatei MDSPROTO hinterlegt sind.

Für die Prozedur MDS_DatumDrehen kann genau der gleiche Prototyp wie für die Main-Procedure in MDMCVTDAT1 verwendet werden, lediglich die Aufrufnamen müssen ausgetauscht werden. Das Gleiche gilt für die Prototypen der Main-Procedure MDMCVTDAT2 und der Prozedur MDS_DatumAufbereiten.

Im folgenden Beispiel wurde der Prototyp für die Prozedur MDS_DatumAufbereiten im Free-Format hinterlegt. Im Free-Format beginnt ein Prototyp immer mit dem Operations-Code DCL-PR und endet mit dem Operations-Code END-PR. Die einzelnen Parameter werden zwischen die beiden Operations-Codes eingefügt. Die Angabe des Operations-Codes DCL-PARM ist bei der Definition von Parameterfeldern sowohl im Prototyp als auch im Procedure Interface optional. Im Gegensatz zur Fix-Format-Codierung muss im Free-Format ein Parametername angegeben werden. Für unbenannte Parameter kann der Sonderwert *N verwendet werden. Die Datentypen und -längen werden im Free-Format in Form von speziellen Datentyp-Schlüsselworten hinterlegt.

3.1.7
Seite 44

```
 ****************************************************************
 * Procedure name: MDS_DatumDrehen
 ****************************************************************
D MDS_DatumDrehen...
D                 PR
D  POutNumDate                  8P 0
D  PInNumDate                   8P 0 CONST
D  PInFmt                       4A   CONST OPTIONS(*NOPASS)

 ****************************************************************
 * Procedure name: MDS_DatumAufbereiten
 ****************************************************************
  DCL-PR MDS_DatumAufbereiten;
         *N    Packed(8) Const;
         *N    Char(4)   Const;
         *N    Char(1)   Const;
      ParOut01 Char(10);
  End-PR;
```

Prototypen für die Prozeduren MDS_DrehenDatum und MDS_AufbereitenDatum

Das Modul kann mit dem folgenden CL-Befehl erstellt werden:

```
CRTRPGMOD MODULE(YOURLIB/MDSCVTDAT)
          SRCFILE(YOURSRCLIB/YOURSRCFIL)
          SRCMBR(MDSCVTDAT)
```

Erstellen des Moduls MDSCVTDAT1

Da für die Prozeduren die gleichen Prototypen wie für die Main-Procedures in den Modulen Verwendung finden – nur mit geändertem Aufrufnamen – muss im aufrufenden Programm außer dem Austausch der Aufrufnamen keine weitere Änderung erfolgen, wie das folgende Source-Beispiel zeigt.

```
 D/Copy QILESRC,MDSProto

 D DatumA          S             10A
 D DatumJJMT       S              8P 0
 D DatumDMYY       S              8P 0
 D DatFmtA         S              4A
 D DatSep          S              1A
 *---------------------------------------------------------------
  /Free
    *INLR = *ON;

    Clear DatumJJMT;
    DatumDMYY = %Dec(%Date(): *EUR);
    MDS_DatumDrehen(DatumJJMT: DatumDMYY: 'TMJJ');
    Dsply DatumJJMT;

    MDS_DatumDrehen(DatumDMYY: 20140701);
    Dsply DatumDMYY;

    DatFmtA= 'TMJJ';
    DatSep = '-';
    MDS_DatumAufbereiten(DatumJJMT: DatFmtA: DatSep: DatumA);
    Dsply DatumA;

    Clear DatumA;
    MDS_DatumAufbereiten(20140701: 'JJMT': '/': DatumA);
    Dsply DatumA;

    Return;
  /End-Free
```

Aufruf der aus dem Modul exportierten Prozeduren

Das Modul kann mit dem folgenden CL-Befehl erstellt werden:

```
CRTRPGMOD MODULE(YOURLIB/MDSCALL)
          SRCFILE(YOURSRCLIB/YOURSRCFIL)
          SRCMBR(MDSCALL)
```

Erstellen Modul MDSCALL – Aufruf von Sub-Prozeduren in Modul MDSCVTDAT

3.1.7
Seite 46

Nachdem beide Module erstellt wurden, können sie mit dem folgenden CL-Befehl zu einem Programm gebunden werden:

```
CRTPGM PGM(YOURLIB/MDSCALL)
       MODULE(MDSCALL MDSCVTDAT )
       ENTMOD(*FIRST)
```

Erstellen des Programms MDSCALL – Aufruf exportierter Prozeduren in Modul

3.1.7.3.3 Von der Sub-Procedure zur Funktion

Im vorangegangenen Beispiel wurden Sub-Prozeduren definiert, in denen lediglich Ein- und Ausgabeparameter verwendet wurden. Beide Prozeduren haben jeweils einen einzigen Ausgabeparameter.

Anstatt das Ergebnis über einen Ausgabeparameter an die rufende Prozedur zurückzugeben, können in ILE auch Rückgabewerte definiert werden.

Funktionen, also Prozeduren mit einem Rückgabewert, werden nicht über den Operations-Code CALLP aufgerufen, sondern genau wie die in RPG integrierten Built-In-Functions. Funktionen können somit in einem EVAL, IF, SELECT, DOU, DOW etc. im erweiterten Faktor 2 angegeben werden. Ebenso ist es möglich ILE-Funktionen zu verschachteln, das heißt, der Rückgabewert aus einer Funktion wird als Input-Only-Parameter an eine andere Built-In- oder ILE-Funktion oder ILE-Prozedur übergeben.

Der Rückgabewert einer Funktion wird im fixen Format beim Procedure Interface in der PI-Zeile und beim Prototypen in der PR-Zeile durch die Angabe eines Datentyps und ggf. der Länge oder eines LIKE-Schlüsselworts (zum Beispiel LIKE, LIKEDS) festgelegt. Im Free-Format erfolgt die Definition des Rückgabewerts im Procedure-Interface in Verbindung mit dem DCL-PI Statement, während die Definition des Rückgabewert für den Prototyp im DCL-PR-Statement erfolgt. Im Free-Format wird der Datentyp durch ein entsprechendes Datentyp-Schlüsselwort angegeben. Im fixen wie auch im Free-Format-Format können die Rückgabewerte anstatt über einen Datentyp auch durch ein LIKE-Schlüsselwort definiert werden.

Der Rückgabewert wird im Faktor 2 des RETURN-Statements an die rufende Prozedur zurückgegeben.

Im folgenden Beispiel wurden die beiden Funktionen MDS_DatumDrehen und MDS_DatumAufbereiten in Funktionen konvertiert. Der Ausgabeparameter wurde jeweils als Rückgabewert definiert.

Bei der Funktion MDF_DatumDrehen wurde der Ausgabeparameter (der erste Parameter in Prozedur MDS_DatumDrehen) als Rückgabewert in der PI-Zeile definiert und aus der Parameterliste entfernt (1).

Sofern kein Datum übergeben wurde, wird der Wert Null (*Zeros) zurückgegeben (2).

Für den Rückgabewert wurde keine separate Variable definiert, stattdessen wird der entsprechende Source Code im erweiterten Faktor 2 des RETURN-Statements angegeben (3).

3.1.7
Seite 48

Bei der Funktion MDF_DatumAufbereiten wird der Ausgabeparameter (letzter Parameter in Prozedur MDS_AufbereitenDatum) als Rückgabewert definiert und aus der Parameterliste entfernt.

In dieser Funktion wird zwar die Variable RTNDatumA (4) definiert, um Ergebnisse zwischenzuspeichern, ein Teil des Source Codes wird jedoch auch im erweiterten Faktor 2 des RETURN-Statements (5) hinterlegt.

```
H NoMain
  ****************************************************************
  /Copy QILESRC,MDFProto
  ****************************************************************
  * Procedure name: MDF_DatumDrehen
 P MDF_DatumDrehen...
 P                 B                   Export

 D MDF_DatumDrehen...
 D                 PI             8P 0                      (1)
 D  PInNumDate                    8P 0 CONST
 D  PInFmt                        4A   CONST OPTIONS(*NOPASS)

 D ParFmt          S              4A   inz('JJMT')
 D*----------------------------------------------------------------
  /FREE
    Monitor;
      If PInNumDate = *Zeros;
         Return *Zeros;                                        (2)
      EndIf;

      If %Parms >= %ParmNum(PInFmt) and PInFmt <> *Blanks;
         ParFmt = PInFmt;
      EndIf;

      If ParFmt = 'TMJJ';
         Return %Dec(%Date(PInNumDate: *EUR): *ISO);           (3)
      Else;
         Return %Dec(%Date(PInNumDate: *ISO): *EUR);           (3)
      EndIf;

    On-Error;
      Return 99991231;
    EndMon;
```

```
 /End-Free
P MDF_DatumDrehen...
P                 E

 ****************************************************************
 * Procedure name: MDF_DatumAufbereiten
 ****************************************************************
  Dcl-Proc MDF_DatumAufbereiten Export;

  DCL-PI *N Char(10);
      ParDateYYMD Packed(8)   Const;
      ParDatFmtA  Char(4)     Const;
      ParDatSep   Char(1)     Const;
  End-PI;

  DCL-S RtnDatumA Char(10);                                    (4)
  DCL-S Datum     Date;
  //-------------------------------------------------------------
    Monitor;

      Datum = %Date(ParDateYYMD: *ISO);

      Select;
      When ParDatFmtA = 'JJMT';
           RtnDatumA   = %Char(Datum: *ISO);                   (4)
      When ParDatFmtA = 'TMJJ';
           RtnDatumA   = %Char(Datum: *EUR);
      When ParDatFmtA = 'MTJJ';
           RtnDatumA   = %Char(Datum: *USA);
      When ParDatFmtA = 'JMT';
           RtnDatumA   = %Char(Datum: *YMD);
      When ParDatFmtA = 'TMJ';
           RtnDatumA   = %Char(Datum: *DMY);
      When ParDatFmtA = 'MTJ';
           RtnDatumA   = %Char(Datum: *MDY);
      EndSl;
```

3.1.7
Seite 50

```
      If ParDatFmtA <> 'JJMT';
         Return %Subst(RtnDatumA: 1: 2) + ParDatSep +    (5)
                %Subst(RtnDatumA: 4: 2) + ParDatSep +
                %Subst(RtnDatumA: 7);
      Else;
         Return %Subst(RtnDatumA: 1: 4) + ParDatSep +    (5)
                %Subst(RtnDatumA: 6: 2) + ParDatSep +
                %Subst(RtnDatumA: 9);
      EndIf;
    On-Error;
      Return *Blanks;
    EndMon;

  End-Proc MDF_DatumAufbereiten;
```

Individuelle Funktionen in einem Modul

Die Prototypen, die in Teildatei MDFPROTO hinterlegt sind, unterscheiden sich von den entsprechenden Procedure Interfaces nur dadurch, dass im fixen Format anstatt der PI-Zeile eine PR-Zeile angegeben wurde und im Free-Format die Operations-Codes DCL-PI und END-PI durch die Operations-Codes DCL-PR und END-PR ersetzt wurden. Während in dem DCL-PI-Statement anstatt des Prozedurnamens der Sonderwert *N verwendet werden kann, ist die Angabe des Prozedurnamens im Prototyp unerlässlich.

```
 ****************************************************************
 * Procedure name: MDF_DatumDrehen
 ****************************************************************
D MDF_DatumDrehen...
D                 PR             8P 0
D  PInNumDate                    8P 0 CONST
D  PInFmt                        4A   CONST OPTIONS(*NOPASS)

 ****************************************************************
 * Procedure name: MDF_DatumAufbereiten
 ****************************************************************
  DCL-PR MDF_DatumAufbereiten Char(10);
      ParDateYYMD Packed(8)   Const;
      ParDatFmtA  Char(4)     Const;
      ParDatSep   Char(1)     Const;
  End-PR;
```

Prototypen für im Modul hinterlegte Funktionen

Das Modul kann über den folgenden CL-Befehl erstellt werden:

```
CRTRPGMOD MODULE(YOURLIB/MDFCVTDAT)
          SRCFILE(YOURSRCLIB/QILESRC)
          SRCMBR(MDFCVTDAT)
```

Erstellen des Moduls MDFCVTDAT

Da Funktionen nicht mit einem CALLP-Statement aufgerufen werden, mussten am rufenden Programm einige Änderungen vorgenommen werden. Die Verwendung von Funktionen eröffnet einige neue, zum Teil verbesserte und vereinfachte Verwendungsmöglichkeiten.

Das folgende Code-Beispiel zeigt das rufende Programm MDFCALL. Wie auch im Programm MDSCALL wird die Variable DatumDMYY mit dem numerischen Tagesdatum im europäischen Format (TTMMJJJJ) gefüllt.

Die Funktion MDF_DatumDrehen hat keinen Ausgabeparameter, sondern gibt das gedrehte Datum in Form eines Rückgabewerts zurück. Die Variable DatumJJMT empfängt den Rückgabewert und wird deshalb auf der linken Seite des Gleichheitszeichens angegeben. Die aufgerufene Funktion MDF_DatumDrehen mit den beiden Input-Only-Parametern wird auf der rechten Seite des Gleicheitszeichens angegeben (1).

Da die Parameterfelder für die Funktion MDF_DatumDrehen durch das Schlüsselwort CONST als Input-Only-Parameter definiert wurden, können anstatt der Parametervariablen Konstanten, Ausdrücke oder fix vorgegebene Werte übergeben werden. Beim zweiten Aufruf der Funktion MDF_DatumDrehen wird im ersten Parameter ein Ausdruck übergeben, in dem ausgehend vom Tagesdatum das Datum in einer Woche berechnet und anschließend in ein numerisches Datum im Format JJJJMMTT konvertiert wird. Beim zweiten Parameter wird das Format JJMT direkt angegeben (2).

Vor dem ersten Aufruf der Funktion MDF_DatumAufbereiten werden zunächst die Variablen DatFmtA und DatSep mit dem gewünschten Ausgabeformat für das aufbereitete Datum sowie mit dem gewünschten Datumstrennzeichen versorgt. Diese Variablen werden beim Aufruf der Funktion MDF_DatumAufbereiten übergeben (3).

Beim letzten Aufruf der Funktion MDF_DatumAufbereiten wird das Ergebnis nicht in eine Variable ausgegeben, sondern direkt mit dem Operations-Code Dsply angezeigt (4). Anstatt für den ersten Parameter einen Wert zu übergeben, wird die Funktion MDF_DatumDrehen aufgerufen, durch die das Datum in Variable DATUMDMYY in das erwartete Format JJJJMMTT konvertiert wird. (5)

3.1.7
Seite 52

```
D/Copy QILESRC,MDFProto

D DatumA          S             10A
D DatumJJMT       S              8P 0
D DatumDMYY       S              8P 0
D DatFmtA         S              4A
D DatSep          S              1A

 *---------------------------------------------------------------
 /Free
   *INLR = *ON;

   Clear DatumJJMT;
   DatumDMYY = %Dec(%Date(): *EUR);
   DatumJJMT = MDF_DatumDrehen(DatumDMYY: 'TMJJ');
(1)
   Dsply DatumJJMT;

   DatumDMYY = MDF_DatumDrehen(%Dec(%Date() + %Days(7)):
      'JJMT');                                                  (2)
   Dsply DatumDMYY;

   DatFmtA= 'TMJJ';
   DatSep = '-';
   DatumA = MDF_DatumAufbereiten(DatumJJMT: DatFmtA:
      DatSep);                                                  (3)
   Dsply DatumA;

   Dsply MDF_DatumAufbereiten(                                  (4)
                    MDF_DatumDrehen(DatumDMYY: 'TMJJ'):         (5)
                    'TMJJ': '-');
   Return;
 /End-Free
```

Prototypen für im Modul hinterlegte Funktionen

Das Modul kann über den folgenden CL-Befehl erstellt werden:

```
CRTRPGMOD MODULE(YOURLIB/MDFCALL)
          SRCFILE(YOURSRCLIB/QILESRC)
          SRCMBR(MDFCALL)
```

Erstellen des Moduls MDFCALL

Nachdem beide Module erstellt wurden, können sie mit dem folgenden CL-Befehl zu einem Programm gebunden werden:

```
CRTPGM PGM(YOURLIB/MDFCALL)
       MODULE(MDFCALL MDSCVTDAT )
       ENTMOD(*FIRST)
```

Erstellen des Programms MDFCALL – Aufruf exportierter Funktionen aus Modul

3.1.7.4 Serviceprogramm

Bislang wurden Programmteile ausgelagert und die ausgelagerten und kompilierten Programmteile/Module by Copy in die eigentlichen Programme eingebunden.

Bei einer modularen Programmierung wird man sehr kleine Programmteile, das heißt nur wenige Code-Zeilen und manchmal sogar nur ein einziges Statement als Prozedur auslagern wollen. Trotz der Vielzahl der sich daraus ergebenden Prozeduren kann die Anzahl der Teildateien/Module überschaubar gehalten werden, indem man mehrere Prozeduren mit ähnlicher Funktionalität (zum Beispiel Datumsfunktionen) in eine Teildatei zusammengruppiert.

Das Problem, das sich beim direkten Binden von Modulen ergibt, ist, dass ein Duplikat des Moduls in jedes Programm kopiert wird. Dies hat wiederum zur Folge, dass

- bei Änderung irgendeiner Prozedur das entsprechende Modul in allen Programmen, in denen es eingebunden ist, ausgetauscht werden sollte. Es ist natürlich auch möglich, das Modul gezielt nur in den Programmen auszutauschen, in denen die betroffene Prozedur aufgerufen wird. Diese Vorgehensweise ist nicht unbedingt zu empfehlen, da sehr leicht ein Programm übersehen werden kann.
- durch Bind by Copy jeweils das komplette Modul kopiert wird, und zwar auch dann, wenn nur eine einzige Prozedur aufgerufen wird. Dadurch multipliziert sich die ursprüngliche Programmgröße unter Umständen um ein Vielfaches, insbesondere dann, wenn aus den Prozeduren wiederum weitere Prozeduren in anderen Modulen aufgerufen werden.

Fazit:

Das direkte Binden von Modulen in Programme ist nicht unbedingt die Ideallösung.

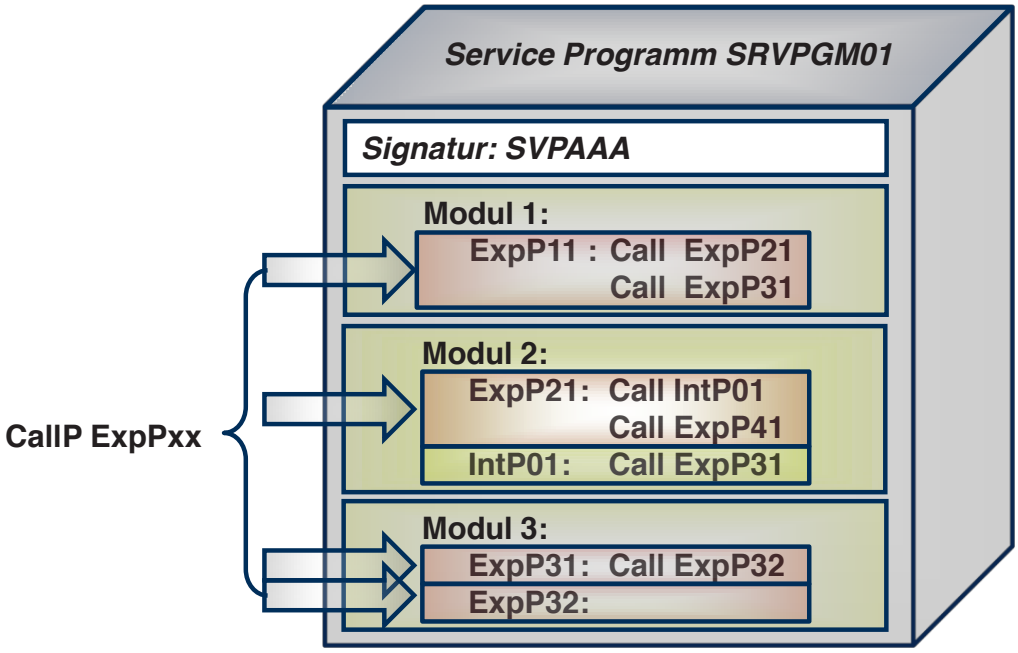
Eine wesentlich bessere Lösung bietet die Verwendung von Serviceprogrammen.

Wie auch Programme können Serviceprogramme aus einem oder mehreren Modulen gebildet werden.

Im Gegensatz zu einem Programm hat ein Serviceprogramm nicht nur einen einzigen Program-Entry-Point (PEP), sondern jede aus einem der gebundenen Module exportierte Prozedur kann von außerhalb des Serviceprogramms aufgerufen werden, also aus anderen Prozeduren in anderen Serviceprogrammen oder anderen Programmen.

Ein Serviceprogramm erhält eine systemweit eindeutige Signatur oder ein Symbol, über die es zusammen mit integrierten Prozeduren und Funktionen lokalisiert werden kann.

Das folgende Beispiel zeigt schematisch den Aufbau des Serviceprogramms SRVPGM01. In diesem Serviceprogramm sind die Module „Modul1“, „Modul2“ und „Modul3“ gebunden. In jedem dieser Module sind exportierte Prozeduren hinterlegt. Die exportierte Prozedur ExpP11 in Modul1 kann ebenso von außen aufgerufen werden wie die exportierte Prozedur ExpP21 in Modul2 oder die exportierten Prozeduren ExpP31 und ExpP32 in Modul3. Bei der Prozedur IntP01 handelt es sich um eine interne Prozedur in Modul2, die nur innerhalb des Moduls2 aufgerufen und ausgeführt werden kann.



Serviceprogramm aus drei Modulen mit internen und exportieren Prozeduren

Da die exportierten Prozeduren aus Serviceprogrammen direkt von außen aufgerufen werden können, ist es nicht notwendig, ein Modul mehrfach in unterschiedliche Programme und/oder Serviceprogramme zu binden. Vielmehr sollte ein Modul nur ein einziges Mal in einem Serviceprogramm gebunden sein.

Anmerkung:

Ein Modul sollte nur ein einziges Mal gebunden werden, entweder in einem Programm oder – sofern Prozeduren aus dem Modul mehrfach verwendet werden können – in einem einzigen Serviceprogramm.

Beim Binden von Serviceprogrammen wird nicht das komplette Objekt (wie im Fall eines Moduls), sondern lediglich die eindeutige Kennung des Serviceprogramms kopiert und integriert. Die Module selber werden nur ein einziges Mal kopiert, und zwar beim Erstellen des Serviceprogramms.

Ändert sich eine Prozedur, so muss das entsprechende Modul nur in dem Serviceprogramm, in dem es eingebunden ist, ausgetauscht werden.

Da die Signatur bei der Änderung von Prozeduren nicht verändert wird, müssen die rufenden (Service-)Programme weder neu erstellt noch anderweitig angepasst werden.

Zum Aktivierungszeitpunkt wird das Serviceprogramm aufgrund der eindeutigen Signatur lokalisiert und in den Speicher geladen.

3.1.7.4.1 Erstellen von Serviceprogrammen

Die Erstellung von ILE-Serviceprogrammen ist immer ein zweistufiger Prozess, das heißt, zunächst müssen alle Module, die in das Serviceprogramm eingebunden werden sollen, mit dem Befehl CRTxxxMOD (Create Module, wobei xxx die Programmiersprache repräsentiert) erstellt werden.

Anmerkung:

Während Programme auch in einem Schritt mit Hilfe des Befehls CRTBNDxxx (xxx Binder-Programm erstellen) generiert werden können, gibt es für Serviceprogramme keine entsprechende Vereinfachung.

Die kompilierten Module werden im Anschluss mit dem CL-Befehl CRTSRVPGM (Serviceprogramm erstellen) zu einem Serviceprogramm gebunden.

Wie beim Erstellen von Programmen werden auch beim Binden von Serviceprogrammen die Module physisch in das Serviceprogramm-Objekt kopiert, das heißt, nach dem Binden werden die Module nicht mehr benötigt und können gelöscht werden.

Anmerkung:

Nach dem Erstellen des Serviceprogramms werden die Module nicht mehr benötigt und können bzw. sollten gelöscht werden.

3.1.7.4.2 Bind by Reference über die Signatur des Serviceprogramms

Prozeduren aus Serviceprogrammen werden ebenso wie Prozeduren aus direkt gebundenen Modulen statisch aufgerufen, das heißt, der Aufruf von Prozeduren ist annähernd so schnell wie der Aufruf von klassischen Subroutinen.

Um einen statischen Prozeduraufruf zu ermöglichen, müssen zum Aktivierungszeitpunkt (also beim Aufruf des rufenden Programms) alle zur Ausführung des Programms notwendigen „Programmteile“ in den Speicher geladen und in Laufbereitschaft versetzt werden.

Module werden physisch in die Programme kopiert (Bind by Copy), wodurch sie bei der Aktivierung des Programms automatisch in den Speicher geladen werden.

Da es sich bei Serviceprogrammen um unabhängige Objekte handelt, können sie de facto nicht automatisch aktiviert werden. Dennoch müssen zum Aktivierungszeitpunkt die benötigten Serviceprogramme lokalisiert, in den Speicher geladen und ebenfalls in Laufbereitschaft versetzt werden.

Um das zu ermöglichen, werden die Serviceprogramme, in denen sich die aufgerufenen Prozeduren befinden, bereits während des Bindens ermittelt. Wird die Prozedur in einem Serviceprogramm gefunden, so wird nicht das Serviceprogramm, sondern nur die eindeutige Signatur des Serviceprogramms in das zu erstellende (Service-)Programm-Objekt integriert (Bind by Reference). Über die in den (Service-)Programm-Objekten gespeicherten eindeutigen Signaturen können die Serviceprogramme zum Aktivierungszeitpunkt lokalisiert werden.

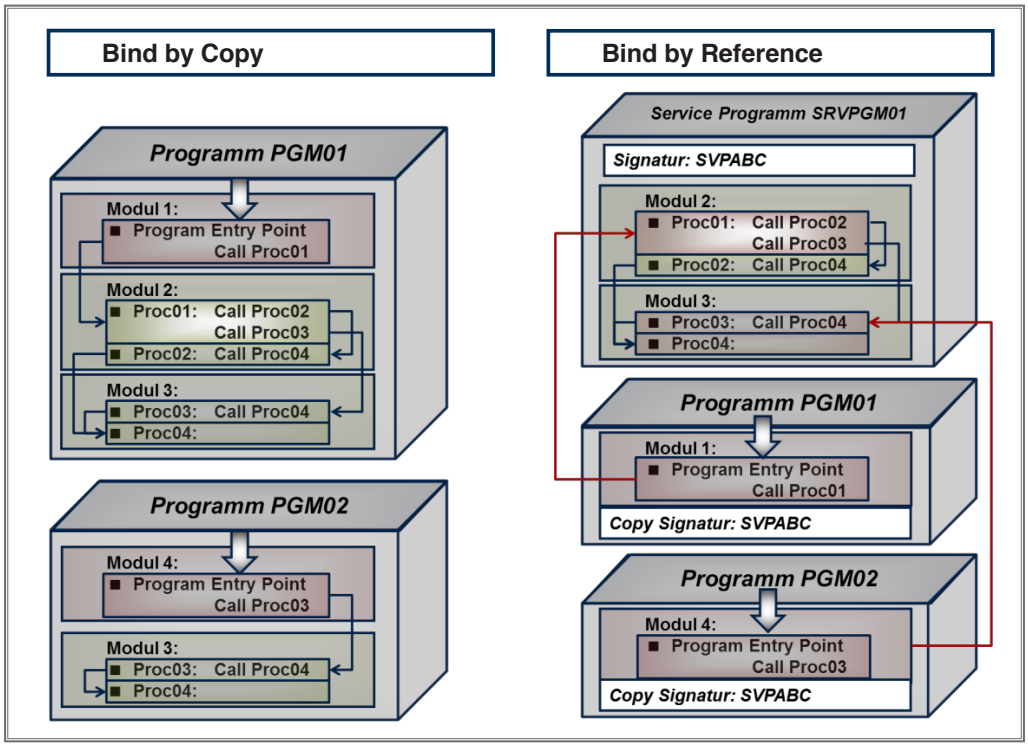
Da lediglich die Signaturen und nicht die kompletten Module in die Programmobjekte integriert werden, muss bei Änderung einer Prozedur lediglich das Modul in dem Serviceprogramm ausgetauscht werden. Das modifizierte Serviceprogramm wird über die unveränderte Signatur lokalisiert.

Das folgende Schaubild zeigt schematisch den Unterschied zwischen Bind by Copy und Bind by Reference.

Beim Bind by Copy auf der linken Seite sind die Module „Modul2“ und „Modul3“ direkt in die Programme PGM01 und PGM02 gebunden, wobei „Modul3“ in beide Programme kopiert wurde. Ändert sich eine Prozedur in Modul3 müssen beide Programme neu gebunden/erstellt werden.

3.1.7
Seite 58

Beim Bind by Reference sind die Module „Modul2“ und „Modul3“ in dem Serviceprogramm SRVPGM01 gebunden. In den Programmen PGM01 und PGM02 werden weiterhin die gleichen Prozeduren aufgerufen. Anstatt jedoch die Module in die Programmobjekte zu kopieren, wird lediglich die Signatur SVPABC des Serviceprogramms SRVPGM01 in die Programme integriert. Ändert sich jetzt eine Prozedur in Modul2 oder Modul3, so muss das Modul lediglich in dem Serviceprogramm SRVPGM01 ausgetauscht werden.



Vergleich – Bind by Copy versus Bind by Reference

3.1.7.4.3 CRTSRVPGM – Serviceprogramm erstellen

Module können mit Hilfe des Befehl CRTSRVPGM zu einem Serviceprogramm gebunden werden.

Dabei muss zunächst der Serviceprogrammname angegeben werden. Der Serviceprogrammname kann völlig frei gewählt werden, unabhängig vom Namen der zu bindenden Modulobjekte.

Die zu bindenden Module müssen aufgelistet werden. Da ein Serviceprogramm keinen eindeutigen Entry-Point hat, ist die Reihenfolge, in der die Module aufgelistet werden, beliebig.

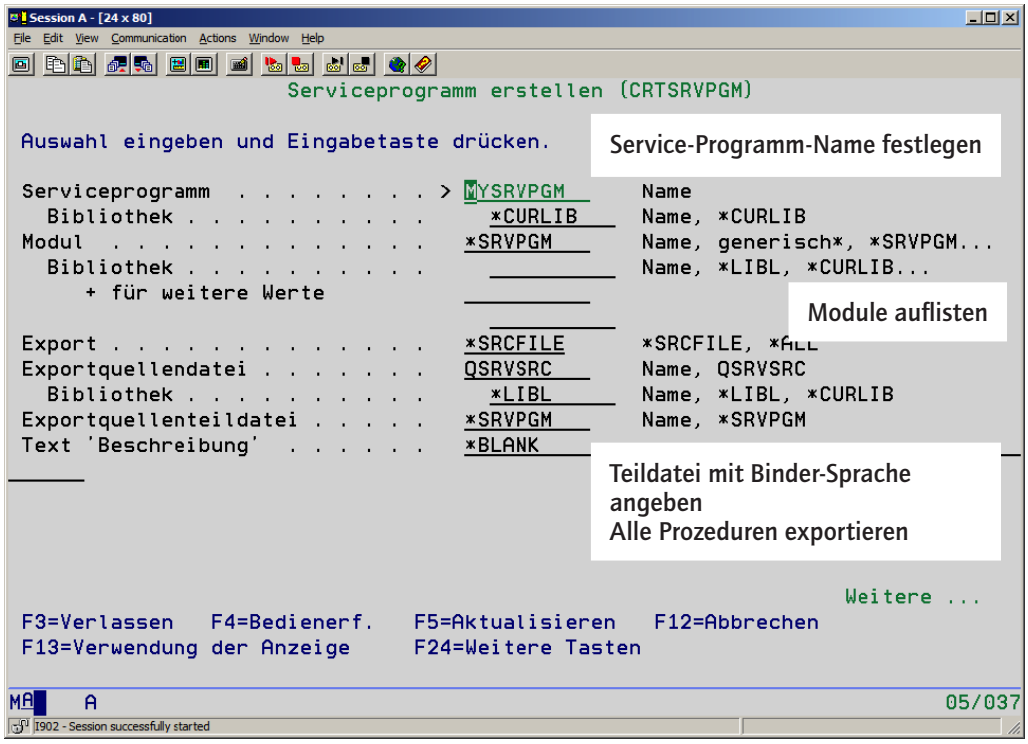
Des Weiteren muss angegeben werden, ob alle exportierten Prozeduren (Schlüsselwort EXPORT) von außerhalb des Serviceprogramms aufgerufen werden können oder nur die in Binder-Sprache hinterlegten Prozeduren in der angegebenen Teildatei exportiert werden dürfen.

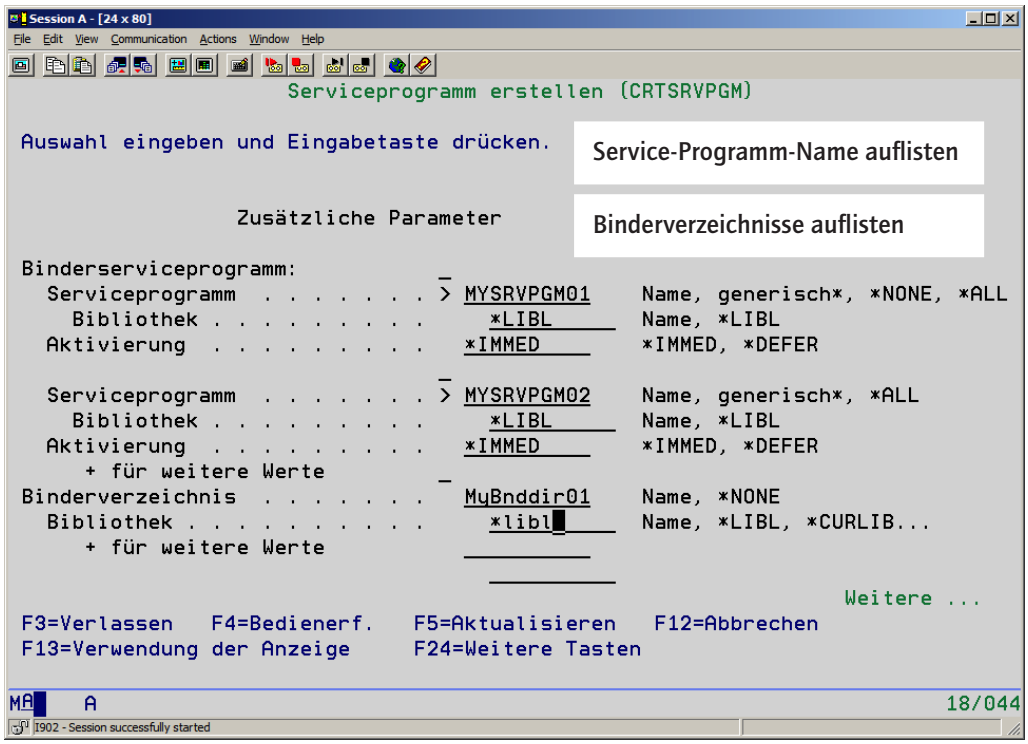
Sofern aus den zu bindenden Modulen Prozeduren aufgerufen werden, die in anderen Serviceprogrammen hinterlegt sind, sind diese Serviceprogramme ebenfalls aufzulisten.

Zur Vereinfachung können Module und/oder Serviceprogramme in Binder-Verzeichnissen aufgelistet werden. In diesem Fall müssen auch die benötigten Binder-Verzeichnisse angegeben werden.

3.1.7
Seite 60

Die folgende Abbildung zeigt den Befehl CRTSRVPGM (Serviceprogramm erstellen):

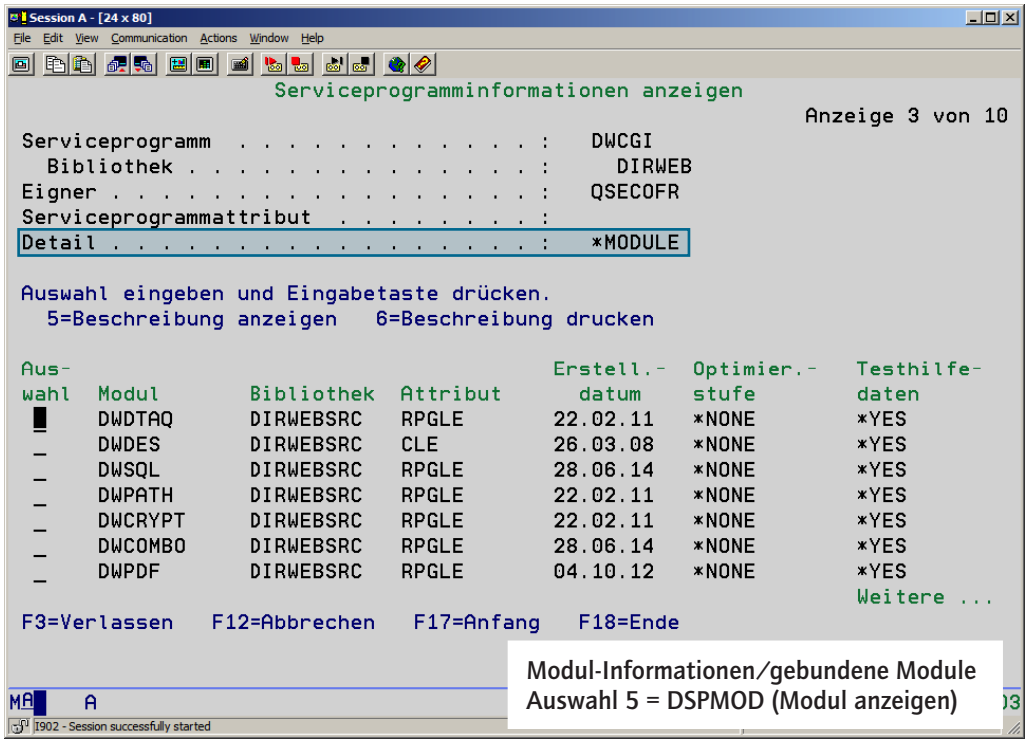




Befehl CRTSRVPGM – Serviceprogramm erstellen

3.1.7.4.4 DSPSRVPGM – Serviceprogramm anzeigen

Mit dem CL-Befehl DSPSRVPGM (Serviceprogramm anzeigen) können Informationen über das Serviceprogramm angezeigt werden.



*DSPSRVPGM – *MODULE – Anzeige der im Serviceprogramm gebundenen Module*

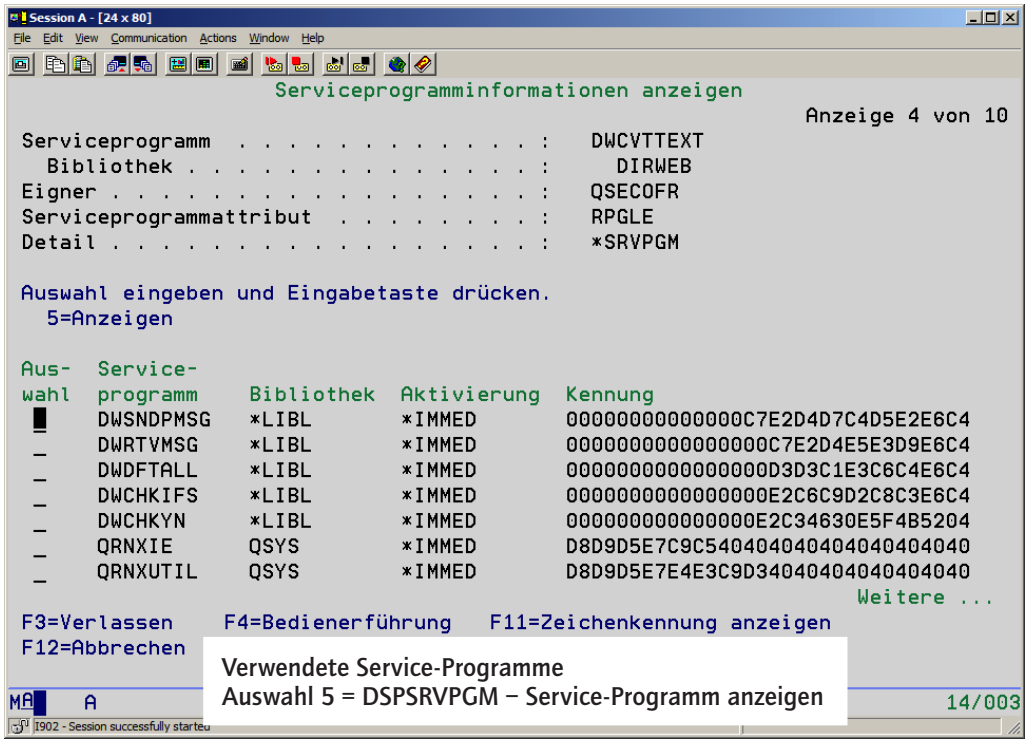
In der folgenden Auflistung werden über die Detailauswahl *SRVPGM in Befehl DSPSRVPGM (Serviceprogramm anzeigen) alle Serviceprogramme aufgelistet, aus denen in Serviceprogramm DWCVTTEXT Prozeduren aufgerufen werden.

Neben anderen Informationen wird an dieser Stelle auch angezeigt, ob das Serviceprogramm direkt bei Aufruf des rufenden Programms (Aktivierung = *IMMED) oder erst bei Aufruf der ersten Prozedur aus dem Serviceprogramm (Aktivierung = *DEFERRED) aktiviert wird.

Des Weiteren ist die eingebundene Signatur der aufgelisteten Serviceprogramme zu sehen.

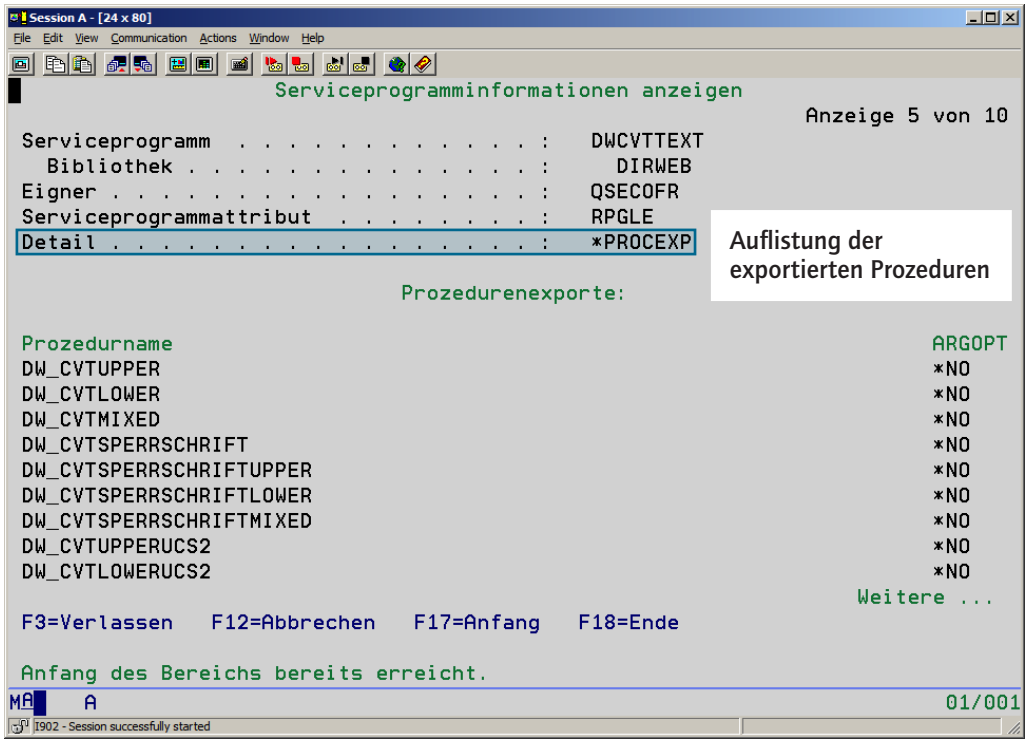
Über Auswahl 5 (= Anzeigen) können mit Hilfe des Befehls DSPSRVPGM (Serviceprogramm anzeigen) detaillierte Informationen über das ausgewählte Serviceprogramm angezeigt werden.

3.1.7
Seite 62



*DSPSRVPGM – *SRVPGM – Anzeige der gebundenen Serviceprogramme*

Die folgende Abbildung zeigt die aus Serviceprogramm DWCVTTEXT exportierten Prozeduren über die Detailauswahl *PROCEXP mit Hilfe des Befehls DSPSRVPGM (Serviceprogramm anzeigen).



*DSPSRVPGM – *PROCEXP – Exportierte Prozeduren*

3.1.7.4.5 Vom Modul zum Serviceprogramm

In den vorangegangenen Beispielen wurde das Modul MDFCVTDAT mit den Funktionen MDF_DatumDrehen und MDF_DatumAufbereiten by Copy in das Programmobjekt MDFCALL gebunden. Solange nur ein einziges Programm die Funktionen verwendet, ist diese Vorgehensweise nicht zu beanstanden. Sobald die Funktionen jedoch in mehreren Programmen verwendet werden, sollten die Module nicht mehr direkt gebunden, sondern in einem Serviceprogramm hinterlegt werden.

Der Source Code wie auch die Prototypen bleiben unverändert, lediglich die Erstellung der Objekte ist eine andere.

Um die einzelnen Beispiele auseinanderhalten zu können, wurden die Teildateien und die Funktionen kopiert und umbenannt, der Source Code wurde jedoch nicht verändert.

Zunächst wird aus der Teildatei SVACVTDAT, in der die beiden Funktionen SVA_DatumDrehen und SVA_DatumAufbereiten enthalten sind, ein Modul erstellt. Dazu wird der folgende CL-Befehl verwendet:

```
CRTSRVPGM SRVPGM(HSCOMMON05/SVACVTDAT)
          MODULE(SVACVTDAT)
          EXPORT(*ALL)
```

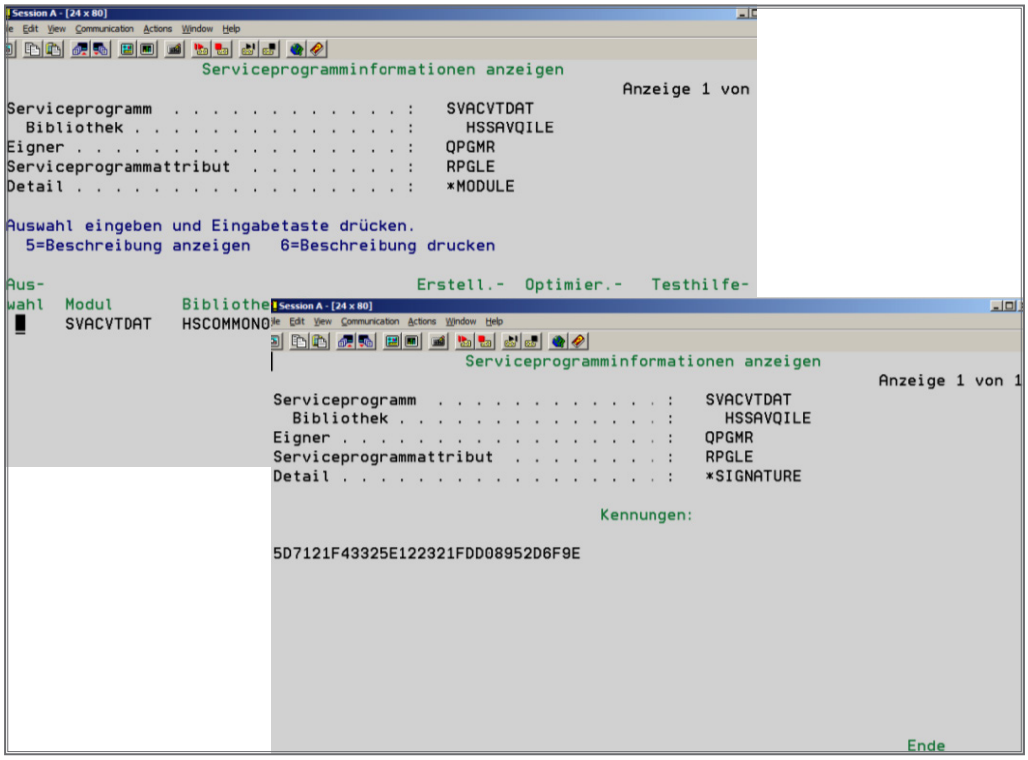
Erstellen des Moduls SVACVTDAT

Das fertige Modul wird jedoch nicht in einem Programm, sondern mit dem folgenden Befehl im Serviceprogramm SVACVTDAT gebunden.

```
CRTPGM PGM(YOURLIB/SVACALL)
       MODULE(SVACALL)
       BNDSRVPGM((SVACVTDAT))
```

Erstellen des Serviceprogramms SVACVTDAT

Die folgende Abbildung zeigt zum einen die Modulinformationen und zum anderen die generierte eindeutige Signatur des Serviceprogramms SVACVTDAT.



DSPSRVPGM – Module und Signatur für Serviceprogramm SVACVTDAT

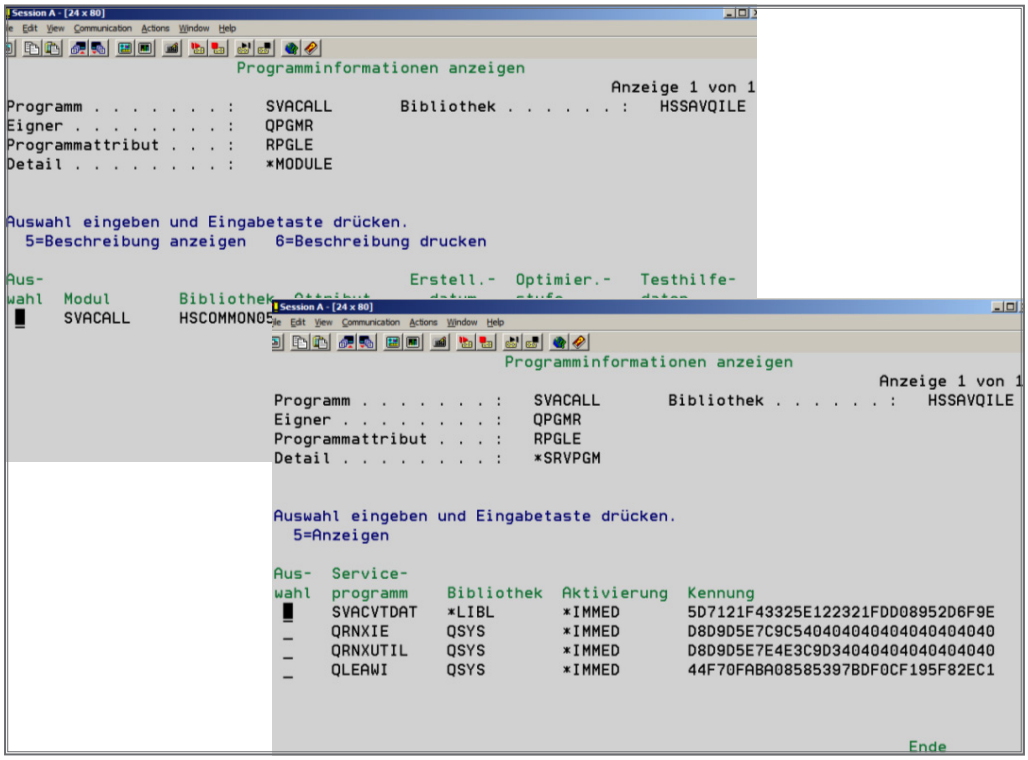
Bei der Umwandlung der rufenden Programme muss jetzt berücksichtigt werden, dass nur noch das Modul mit dem Program-Entry-Point (PEP) in das Programm gebunden wird – also das Modul mit der Main-Procdure, die direkt aufgerufen wird. Die aufgerufenen Prozeduren befinden sich in dem Serviceprogramm SVACVTDAT, das als Binder-Serviceprogramm angegeben werden muss.

Das Programm SVACALL kann mit dem folgenden Befehl erstellt werden:

```
CRTPGM PGM(YOURLIB/SVACALL)
       MODULE(SVACALL)
       BNDSRVPGM((SVACVTDAT))
```

Erstellen des Programms SVACALL

Die folgende Abbildung zeigt einen Auszug der Detailinformationen für die Objekttypen *MODULE und *SRVPGM für das Programm SVACALL aus dem DSPPGM-Befehl. Nur Modul SVACALL ist direkt gebunden. Das Serviceprogramm SVACVDTA ist indes über die Signatur eingebunden.



DSPPGM – Modul- und Serviceprogramm-Informationen für Programm SVACALL

3.1.7
Seite 66

3.1.7.5 Neue Prozeduren in Serviceprogrammen

Im vorangegangenen Beispiel wurden Module mit mehreren Prozeduren, die alle exportiert (Option Export *ALL im CRTSRVPGM-Befehl) wurden, in Serviceprogramme gebunden. Bei einer Änderung von Prozeduren innerhalb der gebundenen Module musste lediglich das entsprechende Modul im zugehörigen Serviceprogramm ausgetauscht werden. Da in den rufenden Programmen und Serviceprogrammen die eindeutige Kennung des Serviceprogramms eingebunden wurde, war weder die Änderung noch das erneute Binden der (Service-)Programme erforderlich.

Wird jedoch zu einem vorhandenen Serviceprogramm eine neue Prozedur oder Funktion hinzugefügt, wird die eindeutige Kennung des Serviceprogramms neu ermittelt und die neue Signatur anstatt der ursprünglichen Signatur in das Serviceprogramm übernommen. Dabei spielt es keine Rolle, ob die neue Prozedur zu einem vorhandenen Modul hinzugefügt wird oder ob ein neues Modul integriert wird.

Da in den rufenden (Service-)Programmen noch die „alte“ Signatur eingebunden ist, müssen alle rufenden (Service-)Programme neu gebunden/erstellt werden. Damit ist das Hinzufügen von neuen Prozeduren zu Serviceprogrammen, die mit der Option EXPORT *ALL erstellt wurden, genau so aufwendig wie das Hinzufügen von Prozeduren zu Modulen, die direkt gebunden werden.

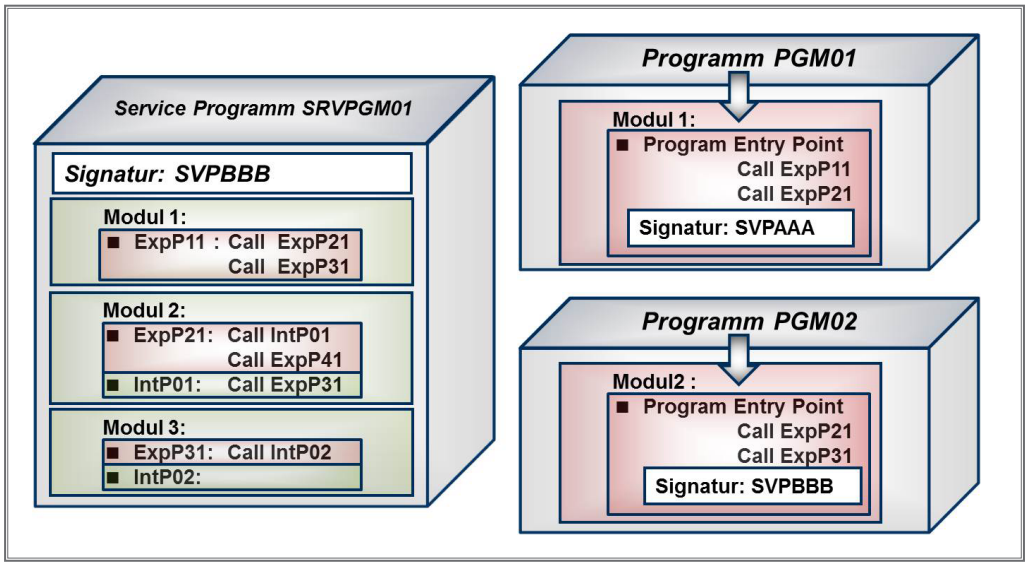
Achtung:

Beim Hinzufügen von neuen Prozeduren/Funktionen zu vorhandenen Serviceprogrammen ändert sich die eindeutige Kennung des Serviceprogramms. Die rufenden (Service-)Programme müssen bei Export *ALL neu gebunden/erstellt werden.

Die folgende Abbildung zeigt das Serviceprogramm SRVPGM01 mit den exportierten Prozeduren ExpP11 und ExpP21 sowie die beiden Programme PGM01 und PGM02.

Aus PGM01 werden die beiden Prozeduren ExpP11 und ExpP21 aus Serviceprogramm SRVPGM01 aufgerufen. In Programm PGM01 wurde die eindeutige Signatur SVPAAA des Serviceprogramms SRVPGM01 eingebunden.

Aus PGM02 wird die Prozedur ExpP21 aus Serviceprogramm SRVPGM01 aufgerufen. Wie auch in Programm PGM01 wurde die eindeutige Signatur SVPAAA des Serviceprograms SRVPGM01 eingebunden.



Aufruf von Prozeduren aus Serviceprogramm

Wird jetzt eine Prozedur in das Serviceprogramm, das mit EXOPRT *ALL erstellt wurde, eingefügt, ändert sich die Signatur. Die (Service-)Programme, aus denen die neue Prozedur aufgerufen wird, werden neu erstellt, wodurch die neue Signatur in diese (Service-)Programme integriert wird. Diese Programme und Prozeduren in den neu erstellten Serviceprogrammen können problemlos ausgeführt werden.

In allen anderen (Service-)Programmen, die Prozeduren aus dem geänderten Serviceprogramm aufrufen, ist jedoch weiterhin die alte Signatur eingebunden. Beim Aufruf von solchen (Service-)Programmen bzw. von Prozeduren aus diesen Serviceprogrammen kann das Serviceprogramm aufgrund der eingebundenen Signatur nicht mehr lokalisiert werden. Dies führt zu Signaturverletzungen, das heißt, das rufende Serviceprogramm kann nicht aktiviert und somit die Prozedur nicht ausgeführt werden.

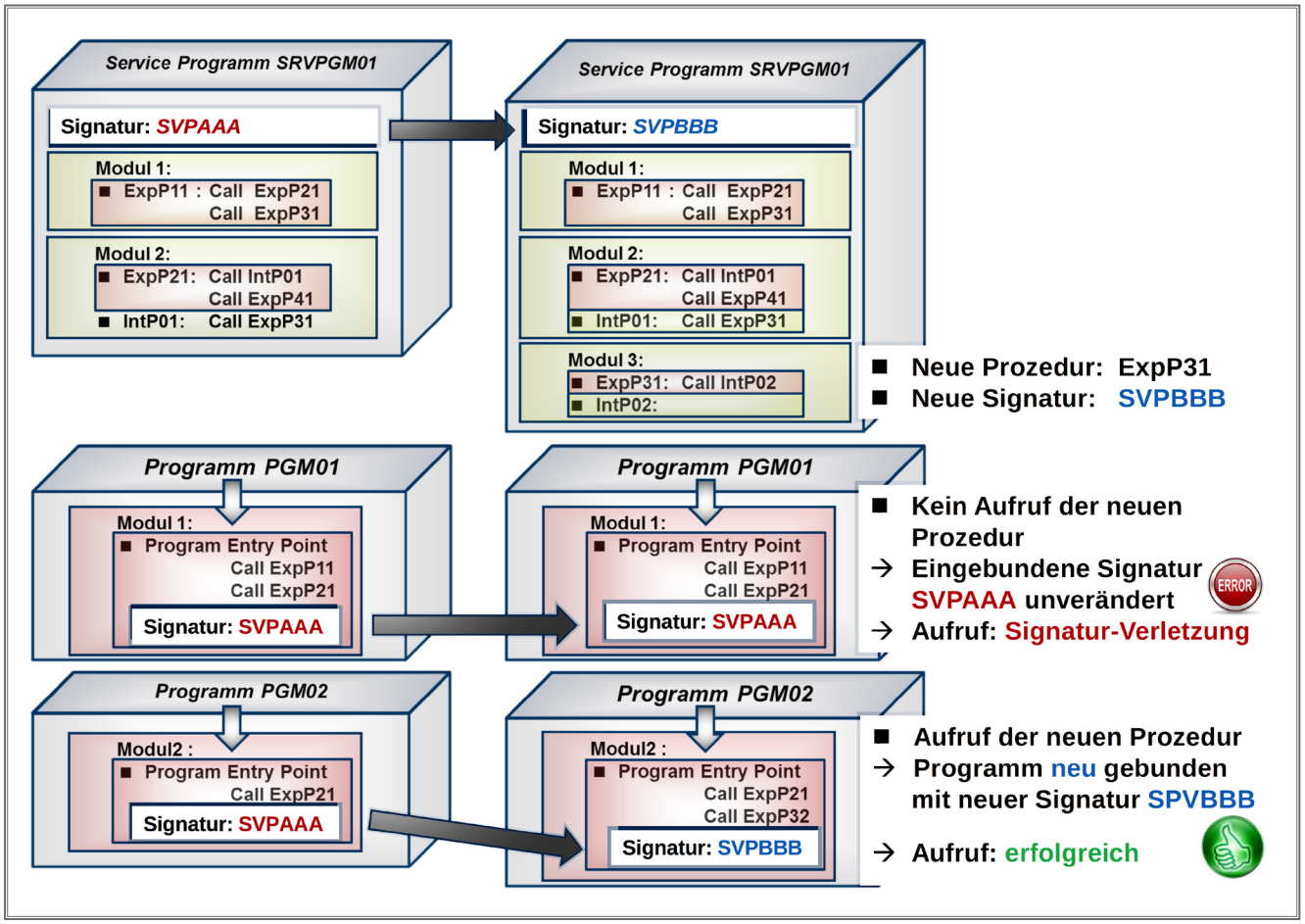
Das folgende Beispiel zeigt die Änderungen, die sich durch das Hinzufügen einer Prozedur in Serviceprogramm SRVPGM01 ergeben und welche Konsequenzen daraus folgen.

Serviceprogramm SRVPGM01 wurde durch das Einbinden des Moduls 3 mit der exportierten Prozedur ExpP31 erweitert. Beim Binden des Serviceprogramms SRVPGM01 wird die Signatur neu ermittelt (SVPBBB). Diese neue Signatur wird im Serviceprogramm hinterlegt und ersetzt die ursprüngliche Signatur SVPAAA.

Da die neue Prozedur in Programm PGM02 benötigt wird, wird das Programm entsprechend geändert und neu erstellt. Damit wird die neue Signatur SVPBBB in das Programm PGM02 integriert.

Da die neue Prozedur nicht in Programm PGM01 benötigt wird, bleibt das Programm unverändert und wird auch nicht neu kompiliert, das heißt, es beinhaltet weiterhin die alte Signatur SVPAAA.

Beim Aufruf des Programms kommt es zu Signaturverletzungen, da das Serviceprogramm SRVPGM01 über die weiterhin hinterlegte „alte“ Signatur nicht mehr lokalisiert werden kann.



*Hinzufügen von Prozeduren in Serviceprogramm mit EXPORT *ALL*

Achtung:

Werden Serviceprogramme ohne Bindersprache mit der Option **EXPORT *ALL** gebunden, müssen beim Hinzufügen von neuen Prozeduren **alle Programme und Serviceprogramme,** aus denen irgendeine Prozedur aus dem erweiterten Serviceprogramm aufgerufen wird, **neu gebunden** werden.

3.1.7
Seite 70

Beispiel: Neue Prozeduren in Serviceprogramm SVACVTDAT

In Serviceprogramm SVACVTDAT werden alle Prozeduren hinterlegt, in denen Datums- und Zeitwerte in irgendeiner Form konvertiert und aufbereitet werden.

Die Firmenleitung hat mitgeteilt, dass zukünftig der Monatsname anstatt des numerischen Monats in allen Datumsangaben auf Dokumenten ausgegeben werden soll. Aus diesem Grund wurde beschlossen, das Serviceprogramm SVACVTDAT um zwei exportierte Prozeduren zu erweitern.

In der ersten Prozedur SVA_Monatsname soll – basierend auf dem übergebenen numerischen Monat – der Monatsname ausgegeben werden. Da die Firma mehrsprachig arbeitet, kann als zweiter optionaler Parameter die Sprache übergeben werden. Aktuell werden nur Deutsch, Englisch und Französisch unterstützt. Sofern die Sprache nicht oder aber eine nicht unterstützte Sprache übergeben wird, wird der Monatsname in Deutsch ausgegeben

Das folgende Beispiel zeigt den Source-Code für diese Funktion. Die Monatsnamen in den unterschiedlichen Sprachen sind lokal in einer verschachtelten Datenstruktur hinterlegt. Sofern ein Monat außerhalb des gültigen Bereichs übergeben wird, wird fix 'Ung.Monat' ausgegeben.

```
Dcl-Proc SVA_MonatsName Export;

 DCL-PI *N VarChar(10);
     ParMonat     Uns(3)    Const;
     ParSprache   Char(1)   Const  Options(*NoPass);
 End-PI;

 DCL-DS DSSprache Qualified;
     DSDeutsch;
       *N  VarChar(10)   Overlay(DSDeutsch)
                         inz('Januar');
       *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                         inz('Februar');
       *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                         inz('März');
       *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                         inz('April');
       *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                         inz('Mai');
       *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                         inz('Juni');
```

```
      *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                        inz('Juli');
      *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                        inz('August');
      *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                        inz('September');
      *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                        inz('Oktober');
      *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                        inz('November');
      *N  VarChar(10)   Overlay(DSDeutsch: *Next)
                        inz('Dezember');
    ArrDeutsch VarChar(10) Overlay(DSDeutsch) Dim(12);
    DSEnglish;
      *N  VarChar(10)   Overlay(DSEnglish)
                        inz('January');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('February');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('March');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('April');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('May');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('June');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('July');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('August');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('September');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('October');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('November');
      *N  VarChar(10)   Overlay(DSEnglish: *Next)
                        inz('December');
```

3.1.7
Seite 72

```
      ArrEnglish VarChar(10) Overlay(DSEnglish) Dim(12);

      DSFrancais;
        *N  VarChar(10)   Overlay(DSFrancais)
                          inz('Janvier');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Fevrier');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Mars');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Avril');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('May');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('June');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Juillet');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Aout');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Septembre');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Octobre');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Novembre');
        *N  VarChar(10)   Overlay(DSFrancais: *Next)
                          inz('Decembre');
      ArrFrancais VarChar(10) Overlay(DSFrancais) Dim(12);
End-Ds;
//-------------------------------------------------------------
      If ParMonat < 1 or ParMonat > 12;
         Return 'Ung.Monat';
      EndIf;

      Select;
      When      %Parms >= %ParmNum(ParSprache)
           and (ParSprache = 'E' or ParSprache = 'e');
           Return DSSprache.ArrEnglish(ParMonat);
```

```
      When      %Parms >= %ParmNum(ParSprache)
           and (ParSprache = 'F' or ParSprache = 'f');
           Return DSSprache.ArrFrancais(ParMonat);
      Other;
           Return DSSprache.ArrDeutsch(ParMonat);
      EndSl;

  End-Proc SVA_MonatsName;
```

Neue Prozedur SVA_MonatsName – Ermitteln des Monatsnamens

Die zweite Funktion, die hinzugefügt werden soll, ist SVA_DatumMitMonat. In dieser Funktion wird das übergebene numerische Datum mit dem Monatsnamen in der gewünschten Sprache aufbereitet.

Das Datum wird als achtstelliger numerischer Wert im Format JJJJMMTT übergeben. Außerdem werden das Datumsformat und das Datumstrennzeichen nach den gleichen Regeln wie bei Prozedur SVA_AufbereitenDatum als Parameter übergeben. Über einen weiteren optionalen Parameter kann die Sprache, in welcher der Monatsname ausgegeben werden soll, übergeben werden.

Der Monatsname wird innerhalb der Prozedur über die Funktion SVA_MonatsName ermittelt. Tag und Jahr werden aus dem übergebenen numerischen Datum entnommen und anschließend im gewünschten Format mit dem Monatsnamen und den entsprechenden Trennzeichen aufbereitet.

Das folgende Beispiel zeigt den Source-Code für die Funktion SVA_DatumMitMonat.

3.1.7
Seite 74

```
Dcl-Proc SVA_DatumMitMonatsName Export;

 DCL-PI *N VarChar(20);
     ParDateYYMD Packed(8)   Const;
     ParDatFmtA  Char(4)     Const;
     ParDatSep   Char(1)     Const;
     ParSprache  Char(1)     Const  Options(*NoPass);
 End-PI;

 DCL-DS *N;
    DateYYMD     Zoned(8);
      Jahr4      Zoned(4)   Overlay(DateYYMD);
      Jahr4A     Char(4)    Overlay(Jahr4);
      Jahr2A     Char(2)    Overlay(Jahr4: 3);
      Monat      Zoned(2)   Overlay(DateYYMD: *Next);
      Tag        Zoned(2)   Overlay(DateYYMD: *Next);
      TagA       Char(2)    Overlay(Tag);
 End-Ds;

 DCL-S MonatsName VarChar(10);
 //-------------------------------------------------------------
     DateYYMD = ParDateYYMD;

     If %Parms >= %ParmNum(ParSprache);
        MonatsName = SVA_MonatsName(Monat: Parsprache);
     Else;
        MonatsName = SVA_MonatsName(Monat);
     EndIf;

     Select;
     When ParDatFmtA = 'JJMT';
          Return Jahr4A + ParDatSep + MonatsName +
                          ParDatSep + TagA;
     When ParDatFmtA = 'TMJJ';
          Return TagA + ParDatSep   + MonatsName +
                        ParDatSep   + Jahr4A;
     When ParDatFmtA = 'MTJJ';
          Return MonatsName + ParDatSep + TagA   +
                              ParDatSep + Jahr4A;
```

```
     When ParDatFmtA = 'JMT';
          Return Jahr2A + ParDatSep + MonatsName +
                          ParDatSep + TagA;
     When ParDatFmtA = 'TMJ';
          Return TagA + ParDatSep   + MonatsName +
                        ParDatSep   + Jahr2A;
     When ParDatFmtA = 'MTJ';
          Return MonatsName + ParDatSep + TagA   +
                              ParDatSep + Jahr2A;
     EndSl;

     Return *Blanks;

  End-Proc SVA_DatumMitMonatsName;
```

Neue Prozedur SVA_DatumMitMonatsName – Aufbereiten des Datums mit Monatsname

Anmerkung:

Die Anforderung, das Datum mit Monatsnamen aufzubereiten, hätte selbstverständlich auch mit einer einzigen Funktion realisiert werden können.
Da mit Hilfe der ILE-Konzepte Codeduplikate soweit möglich vermieden werden sollten, sollte alles, was in irgendeiner Form „wiederverwendbar" aussieht, als eigenständige exportierte Prozedur/Funktion angelegt werden.
Die Ermittlung des Monatsnamens in unterschiedlichen Sprachen kann sicher an mehreren anderen Stellen in anderem Zusammenhang notwendig sein.

Die Prototypen für beide Funktionen werden wie auch die Prototypen der anderen Funktionen im Serviceprogramm SVACVTDAT in Teildatei SVAPROTO hinterlegt.

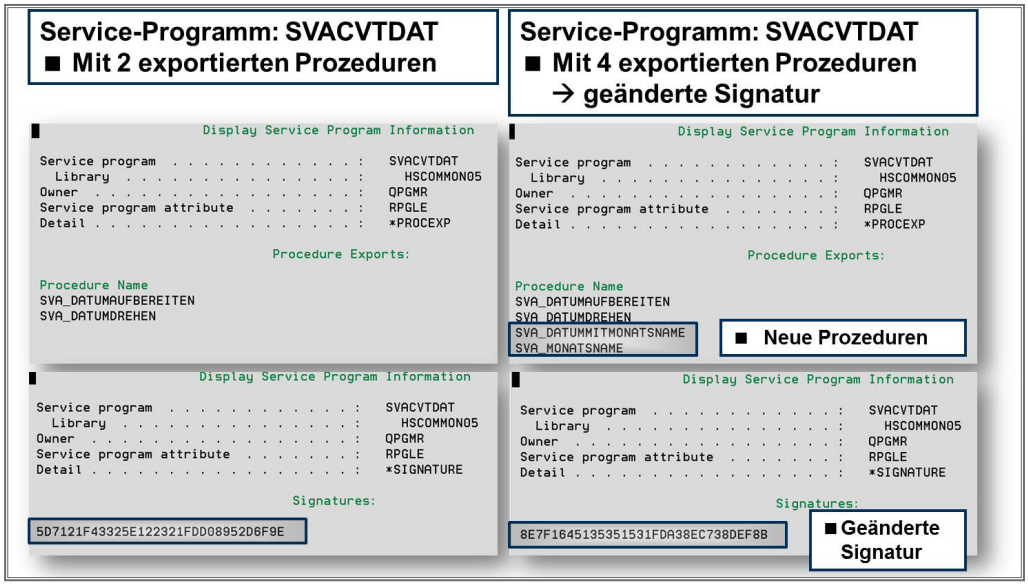
Nach dem Hinzufügen der beiden Funktionen können zunächst das Modul und anschließend das Serviceprogramm mit den folgenden CL-Befehlen neu erstellt werden.

3.1.7
Seite 76

```
CRTRPGMOD MODULE(YOURLIB/SVACVTDAT)
SRCFILE(YOURSRCLIB/QILESRC)
SRCMBR(SVACVTDAT)

CRTSRVPGM SRVPGM(HSCOMMON05/SVACVTDAT)
MODULE(SVACVTDAT)
EXPORT(*ALL)
```

Erstellen Modul und Serviceprogramm SVACVTDAT

Vergleicht man die exportierten Prozeduren und die Signatur vor und nach dem erneuten Binden, so stellt man fest, dass die zusätzlichen Prozeduren in das Serviceprogramm integriert wurden, die Signatur sich jedoch geändert hat, wie die folgende Abbildung beweist.



*Signaturänderung beim Binden von Serviceprogrammen mit EXPORT *ALL*

Wird nach dem erneuten Binden des Serviceprogramms ein (Service-)Programm aufgerufen, in dem eine Prozedur aus dem neu gebundenen Serviceprogramm aufgerufen wird, erfolgt ein Programmabbruch mit Fehlermeldung MCH4431:

Ungültige Programmkennung
Ursache . . . . : Das Quellenprogramm &1 gibt die Kennung &3 an, die von Serviceprogramm &2 nicht unterstützt wird.
Fehlerbeseitigung: Die Serviceprogrammschnittstelle hat sich geändert. Das Quellenprogramm &1 erneut binden.

Nach dem erneuten Binden können die rufenden Programme wieder problemlos aufgerufen werden, da jetzt die neue Signatur eingebunden ist und das Serviceprogramm wieder über diese Signatur lokalisiert werden kann.

Fazit:

Das Binden von Serviceprogrammen mit der Option Export *ALL ist nur dann sinnvoll, wenn sich die Anzahl der exportierten Prozeduren eines Serviceprogramms nie oder nur sehr selten ändert.

3.1.7
Seite 78

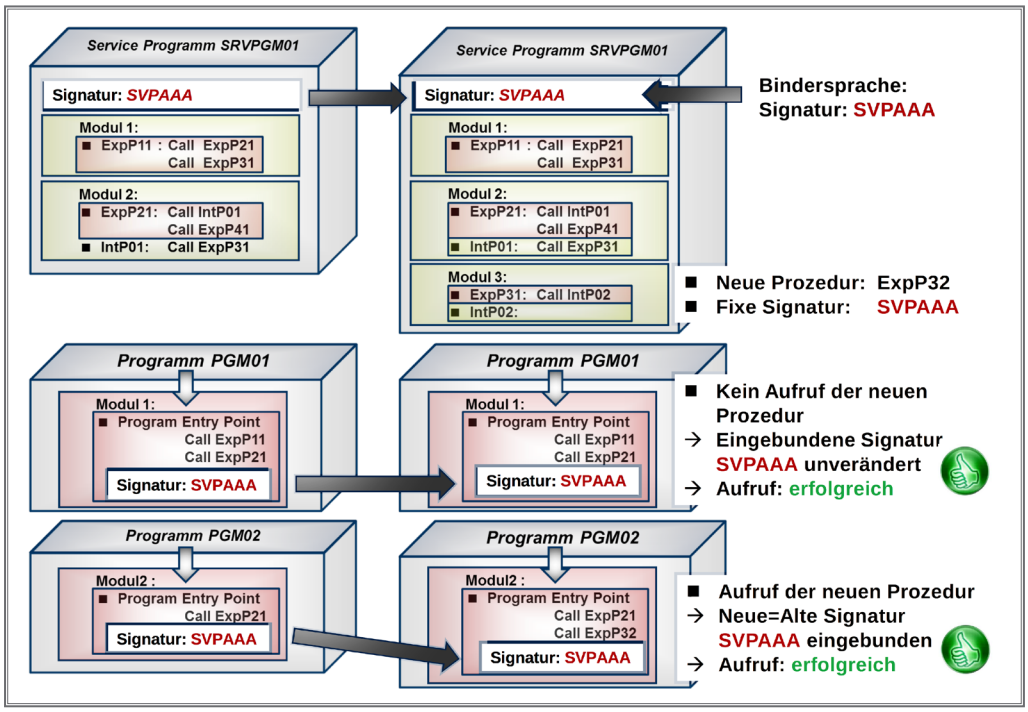
3.1.8 Bindersprache

Das erneute Binden der rufenden (Service-)Programme kann jedoch vermieden werden, wenn die exportierten Prozeduren in Bindersprache in einer eigenen Teildatei hinterlegt werden. Diese Teildatei muss beim Erstellen des Serviceprogramms angegeben werden.

Die eindeutige Signatur des Serviceprogramms wird dann basierend auf den Einträgen in der Binderquelle ermittelt und in das Serviceprogramm eingebunden.

Bei Verwendung der Bindersprache kann die Signatur für das Serviceprogramm fix vorgegeben und damit auch beim Hinzufügen neuer Prozeduren beibehalten werden, wodurch Signaturverletzungen beim Aktivieren vermieden werden. Das Serviceprogramm wird jeweils über die fix vorgegebene Signatur lokalisiert.

Die folgende Abbildung zeigt die gleiche Erweiterung wie im vorangegangenen Beispiel. Dieses Mal wird jedoch durch die Verwendung der Bindersprache die Signatur fix vorgegeben und bleibt damit unverändert. Da die neue Prozedur in PGM02 verwendet wird, muss dieses Programm neu gebunden werden, während das Programm PGM01 unverändert bleibt. In beiden Programmen wird das Serviceprogramm über die vorgegebene Kennung SVPAAA lokalisiert, das heißt, beide Programme können problemlos aufgerufen werden.

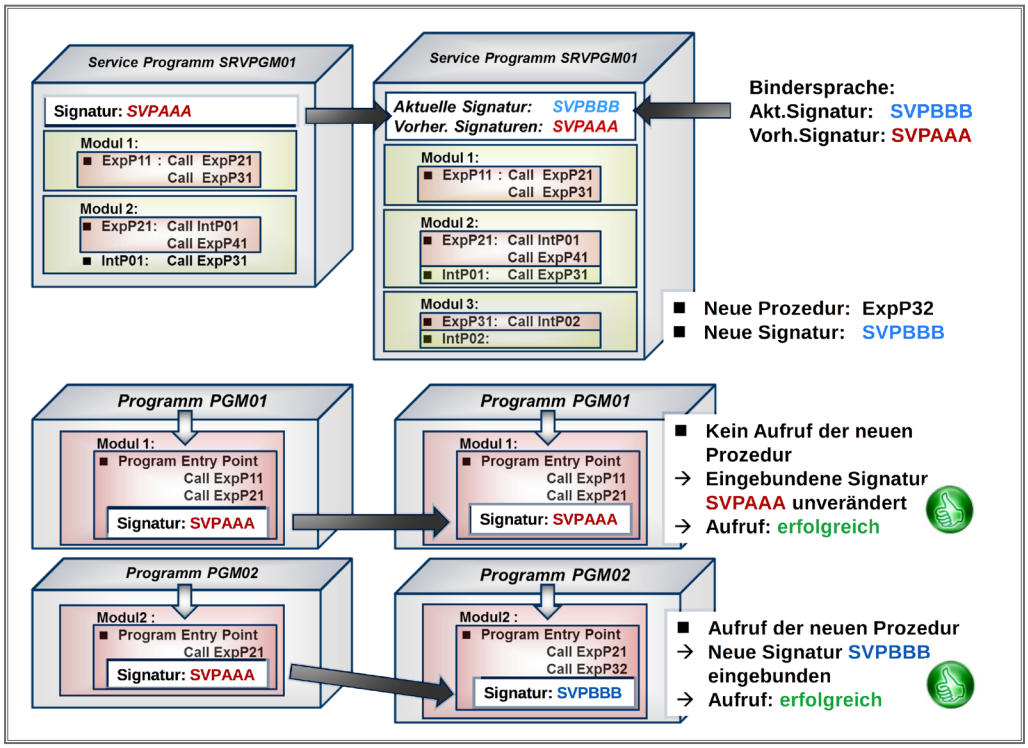


Erweiterung eines Serviceprogramms mit Bindersprache und fixer Signatur

3.1.8
Seite 2

Mit Hilfe der Bindersprache ist es auch möglich, einem Serviceprogramm mehrere Signaturen zuzuweisen, so dass neben der aktuellen Signatur auch die vorhergehenden Signaturen im Serviceprogramm gespeichert werden. Da die vorherigen Signaturen weiterhin im Serviceprogramm hinterlegt sind, ist ein erneutes Binden der rufenden (Service-)Programme nicht erforderlich, da das Serviceprogramm sowohl über die aktuelle als auch die vorhergehenden Signaturen lokalisiert werden kann.

Im folgenden Beispiel werden mit Hilfe der Bindersprache sowohl die aktuelle als auch die vorhergehenden Signaturen in das Serviceprogramm integriert. Da die neue Prozedur aus Programm PGM02 aufgerufen wird, wird dieses neu gebunden, wodurch die aktuelle Signatur SVPBBB eingebunden wird. Das Programm PGM01 bleibt unverändert. Da jedoch beide Signaturen im Serviceprogramm SRVPGM01 hinterlegt sind, kann das Serviceprogramm zur Aktivierungszeit in beiden Fällen lokalisiert werden, das heißt, der Aufruf ist erfolgreich.



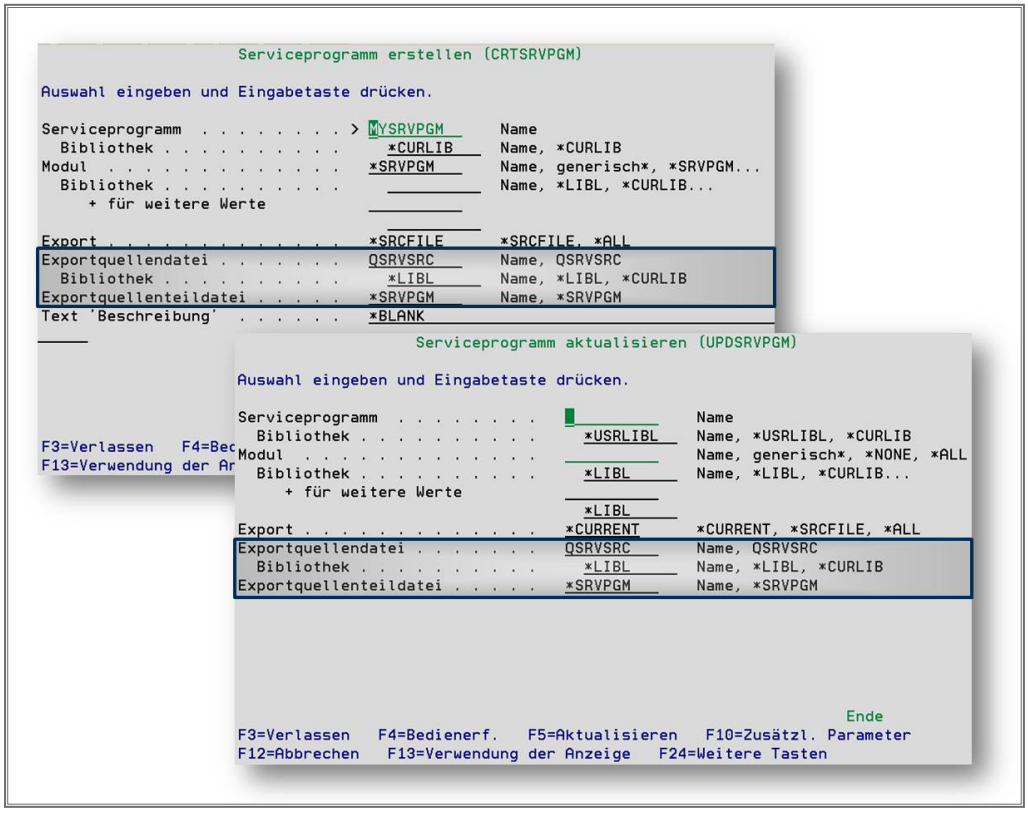
Erweiterung eines Serviceprogramms mit Bindersprache und mehreren Signaturen

3.1.8.1 Erstellen/Aktualisieren eines Serviceprogramms mit Bindersprache

Die Signaturen und exportierten Prozeduren werden in einer eigenen Teildatei der Art BND hinterlegt.

Diese Teildatei kann beim Erstellen oder Aktualisieren des Serviceprogramms mit Hilfe der CL-Befehle CRTSRVPGM (Serviceprogramm erstellen) und UPDSRVPGM (Serviceprogramm aktualisieren) angegeben werden. Pro Serviceprogramm kann nur eine einzige Binderquelle definiert werden.

Die folgende Abbildung zeigt die beiden Befehle CRTSRVPGM (Serviceprogramm erstellen) und UPDSRVPGM (Serviceprogramm aktualisieren). Beim Erstellen des Serviceprogramms kann für die Option EXPORT wahlweise *ALL oder *SRCFILE angegeben werden. Sofern *SRCFILE angegeben wurde, muss eine Binderquelle (Qualifizierte Quellendatei und Teildatei) angegeben werden. Beim Aktualisieren des Serviceprogramms kann für die Option EXPORT noch eine weitere Ausprägung *CURRENT angegeben werden. In diesem Fall bleiben die exportierten Prozeduren und Signaturen unverändert. Dabei spielt es keine Rolle, ob das Serviceprogramm zuvor mit der Exportoption *ALL, das heißt basierend auf allen Prozeduren oder mit Angabe einer Binderquelle erzeugt wurde.



Erstellen/Aktualisieren eines Serviceprogramms mit Bindersprache

3.1.8
Seite 4

Das folgende Code-Beispiel zeigt den Befehl zum Binden des Serviceprogramms MYSRVPGM. Das Serviceprogramm wird aus den Modulen MyMod1 und MyMod2 gebildet. Die Signatur(en) werden über die Informationen in der Binderquelle MYSRVPGM in Quellendatei QSRVSRC (IBM Default) in Quellenbibliothek MYSRCLIB ermittelt und dann in das Serviceprogramm MYSRVPGM integriert.

```
CRTSRVPGM SRVPGM(MYOBJLIB/MYSRVPGM)
MODULE(*LIBL/MYMOD1 MYMOD2)
EXPORT(*SRCFILE)
SRCFILE(MYSRCLIB/QSRVSRC)
SRCMBR(MYSRVPGM)
```

Erstellen eines Serviceprogramms unter Verwendung einer Binderquelle

3.1.8.2 Komponenten der Bindersprache

Bei der Bindersprache handelt es sich um eine eigene Programmiersprache, die jedoch insgesamt nur aus drei Befehlen besteht:

- STRPGMEXP
  Beginn der Auflistung der exportierten Prozeduren/Funktionen
- EXPORT
  Exportierte Prozedur bzw. Funktion
- ENDPGMEXP
  Ende der Auflistung der exportierten Prozeduren/Funktionen

Der Source-Code der Bindersprache wird in Teildateien der Art BND hinterlegt.

Sowohl in RSE als auch in PDM ist eine Bedienerführung für alle Befehle der Bindersprache integriert.

Die exportierten Prozeduren werden immer in einem Block hinterlegt, bestehend aus einem STRPGMEXP-Statement, einem oder mehreren EXPORT-Statements (abhängig von der Anzahl der exportierten Prozeduren) und einem EMDPGMEXP-Statement.

Die Binderquelle muss immer einen Block zur Ermittlung der aktuellen Signatur enthalten. Dieser Block muss an erster Stelle hinterlegt sein. Im Anschluss können mehrere Exportblöcke, die die zuvor exportierten Prozeduren enthalten, angegeben werden.

Sofern die Signatur nicht fix vorgegeben wird, sollten die Blöcke zur Ermittlung der vorherigen Signaturen weder aus der Binderquelle entfernt noch in dieser verändert werden. Fehlt ein solcher Block, kann eine (vorherige) Signatur nicht ermittelt und in das Serviceprogramm übernommen werden. Wurde ein solcher Block verändert, zum Beispiel durch einen neuen Eintrag oder dadurch, dass ein Eintrag gelöscht wurde, wird eine falsche Signatur ermittelt und eingebunden. In solchen Situationen kommt es trotz Verwendung der Bindersprache zu Signaturverletzungen.

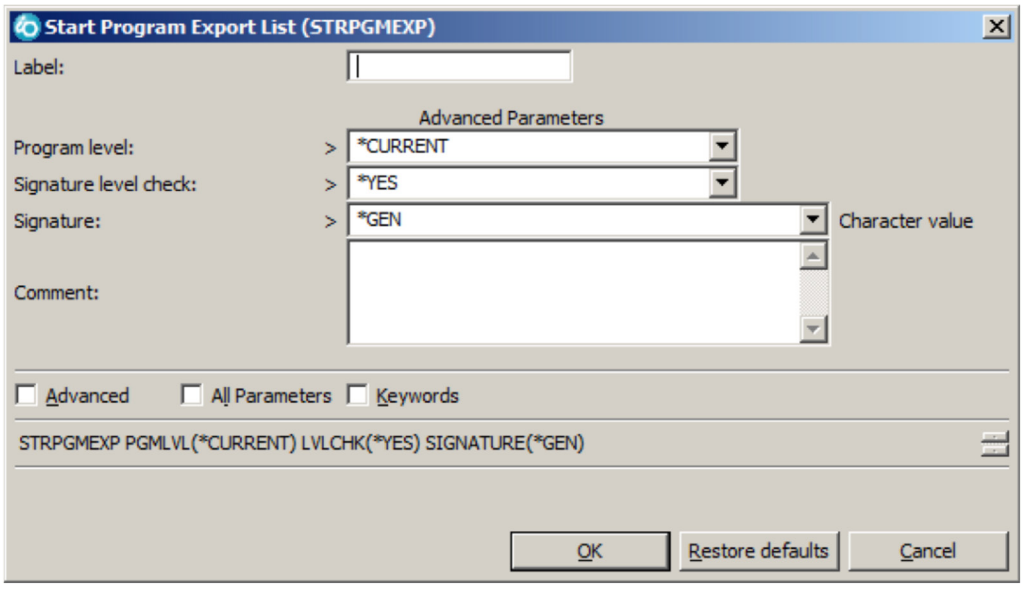
3.1.8.2.1 Befehl STRPGMEXP – Programmexportliste starten

Der Befehl STRPGMEXP kennzeichnet den Beginn der Auflistung der exportierten Prozeduren und Funktionen.

Bei dem Befehl STRPGMEXP können die folgenden Parameter gesetzt werden:

- Programmebene (PGMLVL): kennzeichnet die aktuelle bzw. die vorherige Exportliste
  - *CURRENT aktuelle Exportliste
  - *PRV vorherige Exportliste
- Kennung für Aktualitätsprüfung (LVLCHK)
  - *YES mit Aktualitätsprüfung; Unterlassungswert sollte nicht geändert werden
  - *NO ohne Aktualitätsprüfung; kann bei Änderung der exportierten Prozeduren zu unerwarteten Ergebnissen führen; Aktualitätsprüfung (LVLCHK) *NO sollte vermieden werden
- Kennung (SIGNATURE)
  - *GEN automatische Generierung Unterlassungswert
  - fixe Vorgabe: Die eindeutige Signatur kann fix vorgegeben werden. Erlaubt sind bis zu 16 beliebige Zeichen, die entweder alphanumerisch oder in hexadezimaler Schreibweise angegeben werden können.

Die folgende Abbildung zeigt die Bedienerführung für den Befehl STRPGMEXP in RDi.



Bindersprache – Befehl STRPGMEXP – Beginn des Programmexportblocks

3.1.8.2.2 Bindersprache: Befehl EXPORT

Nach dem STRPGMEXP werden die exportierten Prozeduren mit Hilfe des Befehls EXPORT angegeben.

Der Befehl EXPORT hat nur einen einzigen Parameter:

Exportierter Symbolname (SYMBOL): Name der zu exportierenden Prozedur Wird der Prozedurname direkt angegeben, so wird er unabhängig von der angegebenen Schreibweise intern in Großbuchstaben konvertiert. Damit können alle RPG-Prozeduren und -Funktionen, bei denen das Schlüsselwort EXTPROC nicht im Prototyp angegeben wurde, direkt angegeben werden.

Sofern im Prototyp das Schlüsselwort EXTPROC verwendet und der Prozedurname an dieser Stelle case sensitive angegeben wurde, muss auch der Prozedur- oder Funktionsname in der korrekten Schreibweise entweder in einfache Hochkommata (zum Beispiel 'ProzedurName') oder in Anführungszeichen (zum Beispiel "ProzedurName") eingebettet angegeben werden.

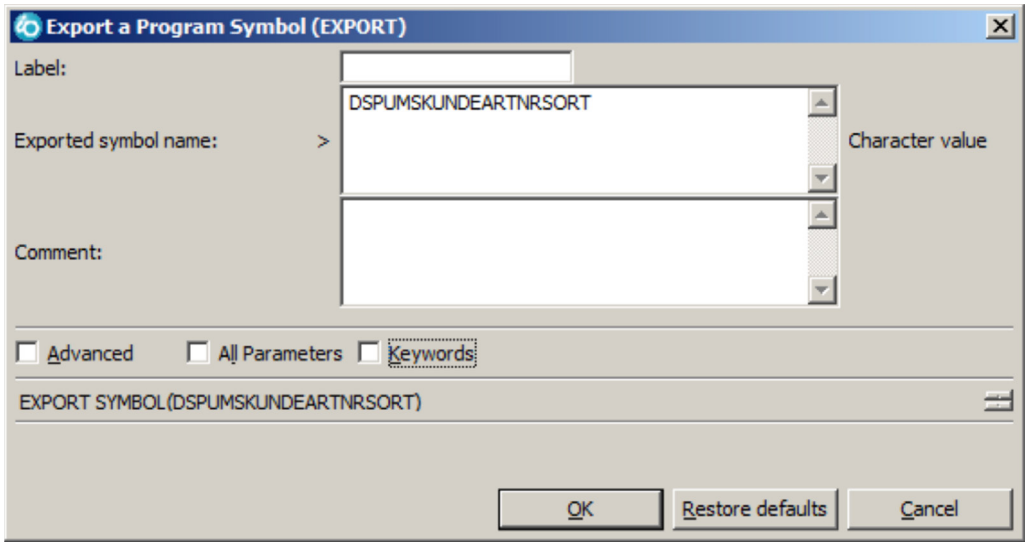
Dies ist insbesondere wichtig beim Anlegen der Bindersprache für Prozeduren und Funktionen, die in anderen Programmiersprachen geschrieben wurden, zum Beispiel in C oder JAVA.

Für jeden Exportblock können 1 bis n EXPORT-Statements angegeben werden.

Wichtig:

Neue exportierte Prozeduren und Funktionen müssen immer am **Ende der Exportliste hinzugefügt** werden, da die Ermittlung der Prozeduren intern nicht über den eigentlichen Prozedurnamen erfolgt, sondern über die Nummer und Reihenfolge des Eintrags in der Exportliste.
Eine veränderte Reihenfolge bewirkt, dass falsche Prozeduren aufgerufen werden, was zumindest zu unerwarteten Ergebnissen und Abbrüchen führt.

Die folgende Abbildung zeigt die Bedienerführung für den Befehl EXPORT in RDi.



Bindersprache – Befehl EXPORT – Prozedurexport

3.1.8.2.3 Bindersprache: Befehl ENDPGMEXP

Der Befehl ENDPGMEXP kennzeichnet das Ende der Exportliste und wird ohne Parameter angegeben.

3.1.8.2.4 Beispiel Binderquelle

Das folgende Beispiel zeigt eine Binderquelle mit einem aktuellen und einem vorherigen Exportblock.

Die Signatur wird in beiden Fällen automatisch ermittelt.

Bei der vorherigen Exportliste wurden drei Prozeduren exportiert. Die Prozedur Cvt2Lower wird in Großbuchstaben konvertiert, da keine Anführungszeichen angegeben wurden. Die beiden anderen Prozeduren sind case sensitiv.

Bei dem aktuellen Exportblock wurden die zuvor exportierten Prozeduren in der gleichen Schreibweise übernommen. Am Ende des Blocks wurden die neuen exportierten Prozeduren TextSpacing und TextSpacingUpper hinzugefügt.

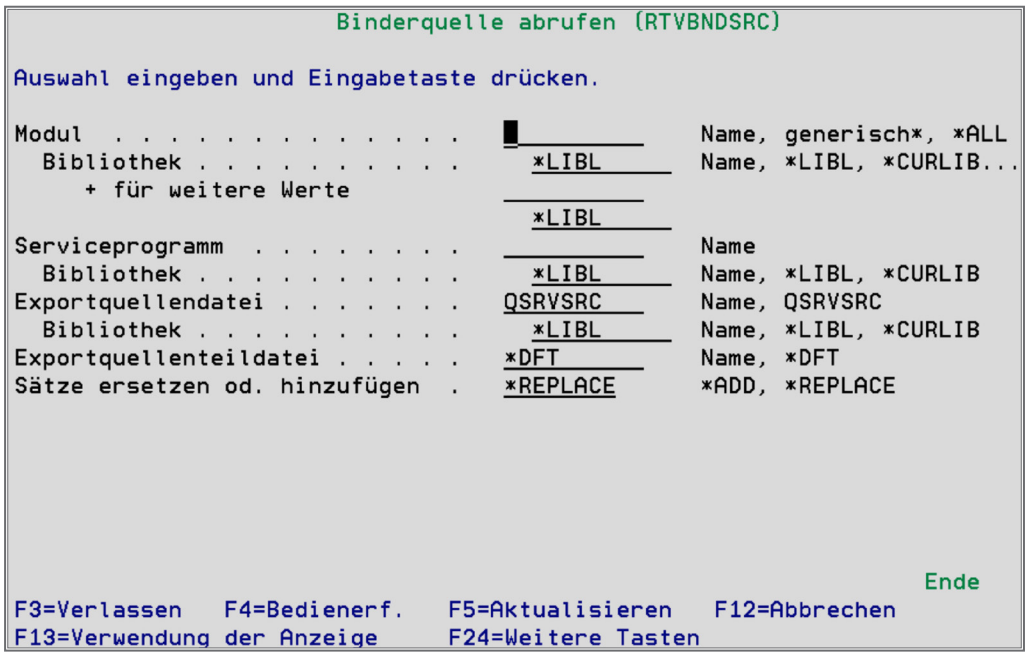
```
STRPGMEXP PGMLVL(*CURRENT)
EXPORT SYMBOL(Cvt2Lower)
EXPORT SYMBOL('Cvt2Upper')
EXPORT SYMBOL("Cvt2Mixed")
EXPORT SYMBOL(TextSpacing)
EXPORT SYMBOL(TextSpacingUpper)
ENDPGMEXP

STRPGMEXP PGMLVL(*PRV)
EXPORT SYMBOL(Cvt2Lower)
EXPORT SYMBOL('Cvt2Upper')
EXPORT SYMBOL("Cvt2Mixed")
ENDPGMEXP
```

Bindersprache

3.1.8.3 CL-Befehl: RTVBNDSRC – Binderquelle abrufen

Mit Hilfe des Befehls RTVBNDSRC (Binderquelle abrufen) können die Exporte und die Signaturen aus einer Gruppe von Modulen, einem Serviceprogramm oder beidem ermittelt und in Bindersprache in eine Teildatei ausgegeben werden.

Die folgende Abbildung zeigt den Befehl RTVBNDSRC (Binderquelle abrufen).



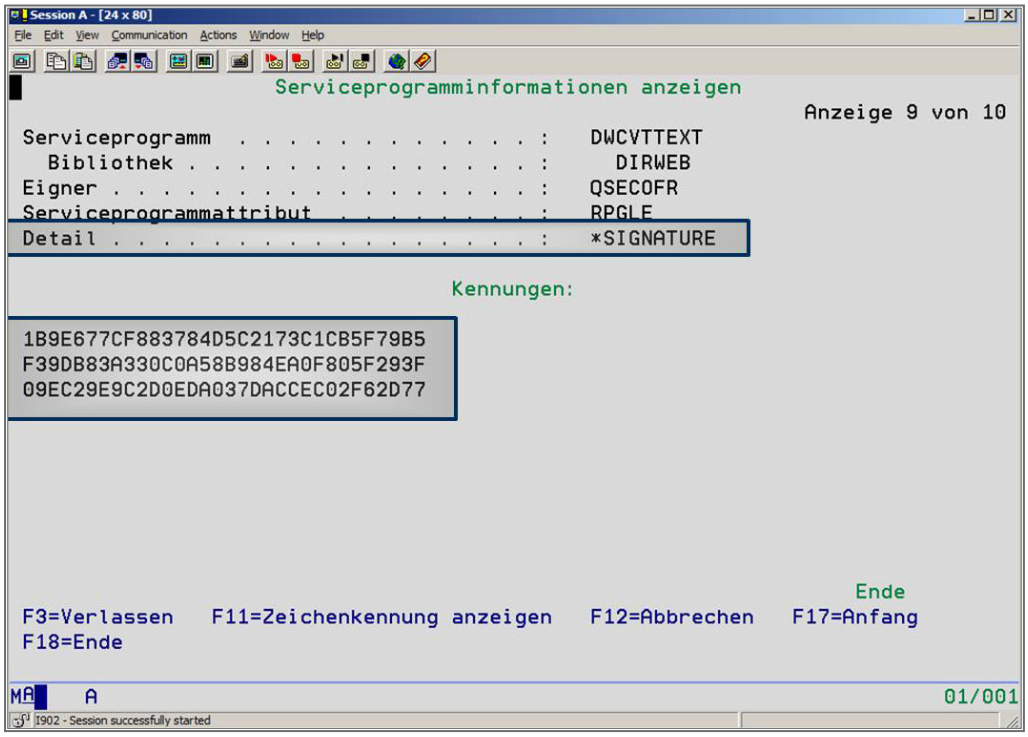
CL-Befehl RTVBNDSRC – Binderquelle abrufen

Mit dem folgenden Befehl wird die Binderquelle für das Serviceprogramm STRING in Bibliothek HSCOMMON05 ermittelt und in die Teildatei STRINGX in Datei QSRVPGMSRC in Bibliothek HSCOMMON05 ausgegeben.

```
RTVBNDSRC SRVPGM(HSCOMMON05/STRING)
SRCFILE(HSCOMMON05/QSRVSRC)
SRCMBR(STRINGX)
```

Erstellen einer Binderquelle für ein Serviceprogramm

In dem Serviceprogramm STRING sind insgesamt drei Signaturen integriert, wie die folgende Abbildung belegt.



Serviceprogramm mit mehreren Signaturen

Die über den CL-Befehl RTVBNDSRC erzeugte Binderquelle wird im nächsten Code-Beispiel gezeigt. Alle im Serviceprogramm hinterlegten Signaturen wurden ermittelt und im Hex-Format in die Binderquelle übernommen. Für jede Signatur wurde ein Exportblock generiert. Prozedurexporte werden jedoch nur für die aktuelle Signatur aufgelistet.

```
STRPGMEXP PGMLVL(*CURRENT) SIGNATURE(X'0CF63EE7EA8E4EA1603C57457572B955')
/*************************************************************/
/* *SRVPGM STRING HSCOMMON05 09/08/14 12:17:06 */
/*************************************************************/
EXPORT SYMBOL("CVT2LOWER")
EXPORT SYMBOL("CVT2UPPER")
EXPORT SYMBOL("CVT2MIXED")
EXPORT SYMBOL("RTVSTRINGPOSINFOCASE")
EXPORT SYMBOL("RTVSTRINGPOSINFO")
EXPORT SYMBOL("TEXTSPACING")
EXPORT SYMBOL("TEXTSPACINGUPPER")
ENDPGMEXP
STRPGMEXP PGMLVL(*PRV) SIGNATURE(X'42333167100A5A2D4C616E1245641FA6')
ENDPGMEXP
STRPGMEXP PGMLVL(*PRV) SIGNATURE(X'000000000000D716F92B0B06F4F704F3')
ENDPGMEXP
```

Mit Befehl RTVBNDSRC generierte Binderquelle

Binderquellen, die mit dem Befehl RTVBNDSRC erzeugt wurden, können jederzeit beim erneuten Binden des Serviceprogramms angegeben werden.

Ebenso ist es möglich, die (neu) generierte Binderquelle manuell zu bearbeiten, das heißt, einen neuen Exportblock oder neue exportierte Prozeduren in den aktuellen Exportblock einzufügen. Die modifizierte Binderquelle kann beim erneuten Binden des Serviceprogramms verwendet werden.

3.1.8.4 Beispiel: Serviceprogramm mit fixer Signatur

In den vorangegangenen Beispielen mussten zu dem Serviceprogramm SVACVTDAT zwei neue Prozeduren hinzugefügt werden. Da das Serviceprogramm jedoch mit der Option EXPORT *ALL erstellt wurde, mussten alle rufenden (Service-)Programme neu erstellt werden, um die geänderte Signatur in die (Service-)Programmobjekte zu integrieren. Anderenfalls kommt es zu Signaturverletzungen.

Wie sieht das Ganze aus, wenn die Bindersprache verwendet wird?

Wir gehen von der gleichen Ausgangssituation aus:

- Das Serviceprogramm SVFCVTDAT mit den exportierten Prozeduren SVF_DatumDrehen und SVF_DatumAufbereiten soll um die Prozeduren SVF_Monatsname und SVF_DatumMitMonatsname erweitert werden.
- Das Serviceprogramm bzw. Prozeduren aus dem Serviceprogramm werden in den Programmen SVFCALL1 und SVFCALL2 aufgerufen.
- Die neuen Prozeduren werden nur in SVFCALL2 benötigt.

Der Source-Code für alle Objekte entspricht dem Source-Code im Beispiel: Neue Prozeduren in Serviceprogramm SVACVTDAT. Um eine klare Unterscheidung zu erhalten, wurden die Teildateien, Prozeduren und Objekte umbenannt (statt des Präfixes SVA wurde überall das Präfix SVF verwendet).

Da ein erneutes Binden der rufenden (Service-)Programme vermieden werden soll, wird mit Bindersprache gearbeitet.

Zunächst wird für das ursprüngliche Serviceprogramm SVFCVTDAT mit den beiden exportierten Prozeduren SVF_DatumDrehen und SVF_DatumAufbereiten die Binderquelle SVFCVTDAT in Datei QSRVSRC wie folgt erstellt. Die Signatur wird in diesem Fall fix mit Signaturname = Serviceprogrammname festgelegt.

```
    STRPGMEXP SIGNATURE('SVFCVTDAT')
     EXPORT   SYMBOL(SVF_DATUMDREHEN)
     EXPORT   SYMBOL(SVF_DATUMAUFBEREITEN)
    ENDPGMEXP
```

Binderquelle mit fixer Signatur

Im Anschluss werden das Modul und das Serviceprogramm SVFCVTDAT mit den folgenden CL-Befehlen erstellt:

```
CRTRPGMOD MODULE(YOURMODLIB/SVFCVTDAT)
SRCFILE(YOURSRCLIB/QILESRC)
SRCMBR(SVAFVTDAT)

CRTSRVPGM SRVPGM(YOURLIB/SVFDAT)
MODULE(YOURMODLIB/SVFDAT)
EXPORT(*SRCFILE)
SRCFILE(YOURSRCLIB/QSRVSRC)
SRCMBR(*SRVPGM)
```

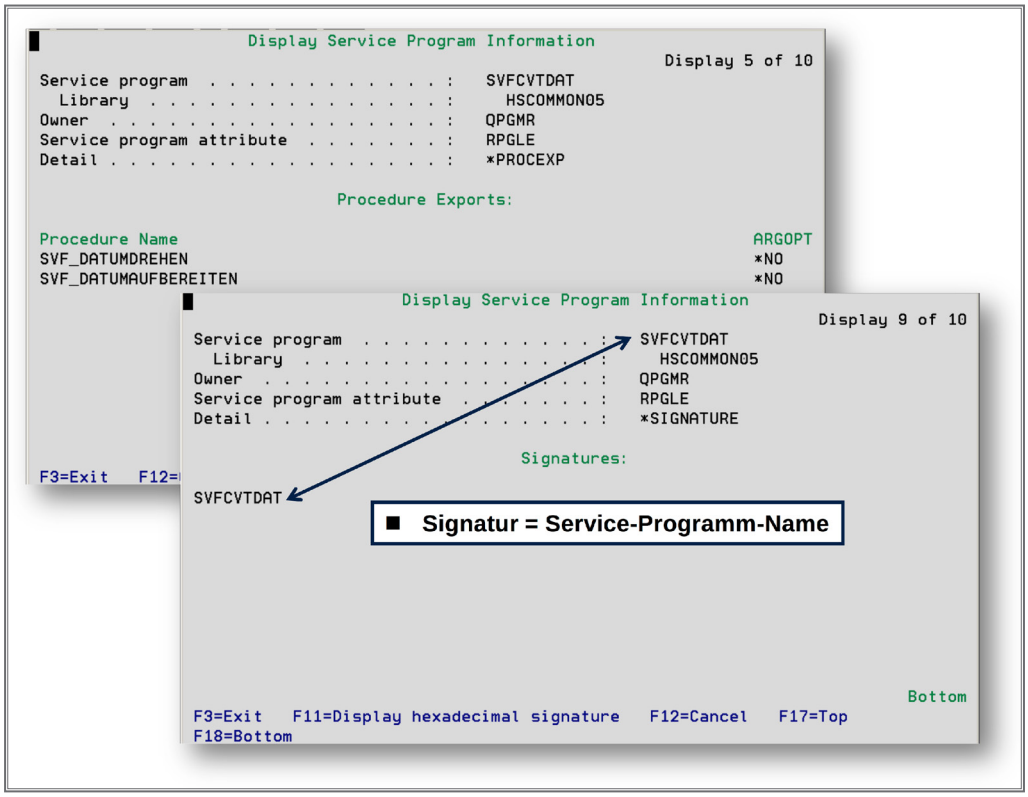
Erstellen Modul und Serviceprogramm mit Bindersprache und fixer Signatur

Anmerkung:

Nach dem Binden sind Module nicht länger erforderlich und können/sollten gelöscht werden.

3.1.8
Seite 14

Die folgende Abbildung zeigt die aus dem Serviceprogramm exportierten Prozeduren sowie die Signatur des Serviceprogramms über den Befehl DSPSRVPGM an.



Exportierte Prozeduren und Signatur aus Serviceprogramm mit fixer Signatur

Die Programme SVFCALL1 und SVFCALL2 werden zweistufig (Modul/Programm) jeweils mit den folgenden Befehlen erstellt:

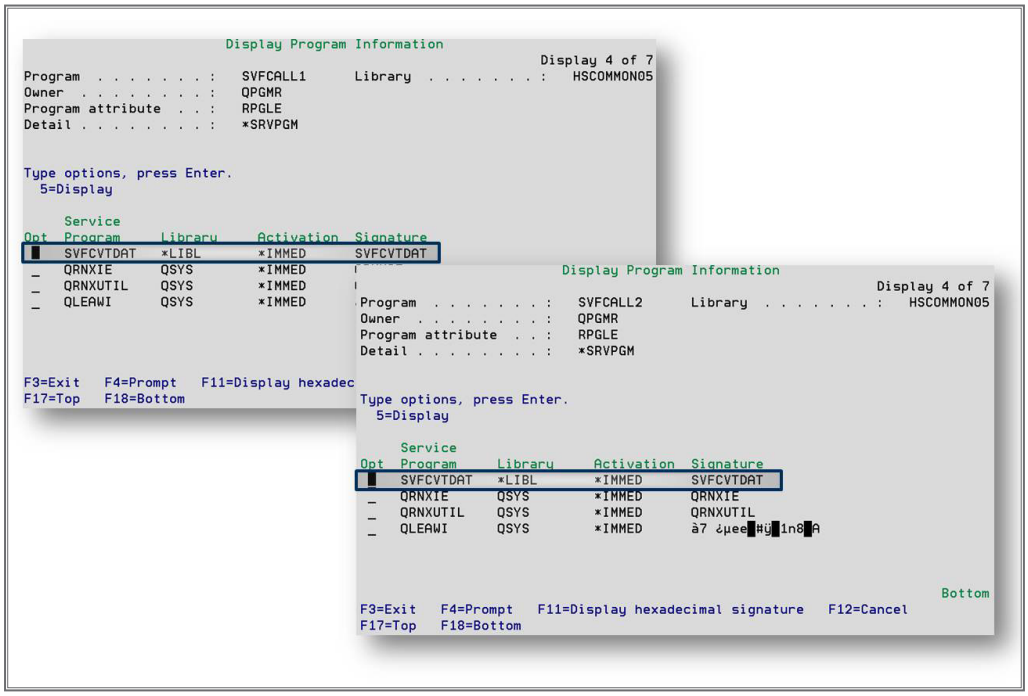
```
CRTRPGMOD MODULE(YOURMODLIB/SVFCALLx)
   SRCFILE(YOURSRCLIB/QILESRC)

CRTPGM PGM(YOURLIB/SVFCALLx)
   MODULE(*PGM)
   BNDSRVPGM((SVFCVTDAT))
```

Erstellen rufender Programme für Serviceprogramm mit fixer Signatur

Anmerkung:

Nach dem Binden sind Module nicht länger erforderlich und können/sollten gelöscht werden.

In beiden Programmen wurde die fix vorgegebene Signatur des Serviceprogramms integriert, wie die folgende Abbildung mit einem Auszug aus dem Befehl DSPPGM (Programmanzeige) belegt.



Einbindung der fixen Signatur des Serviceprogramms in rufende Programme

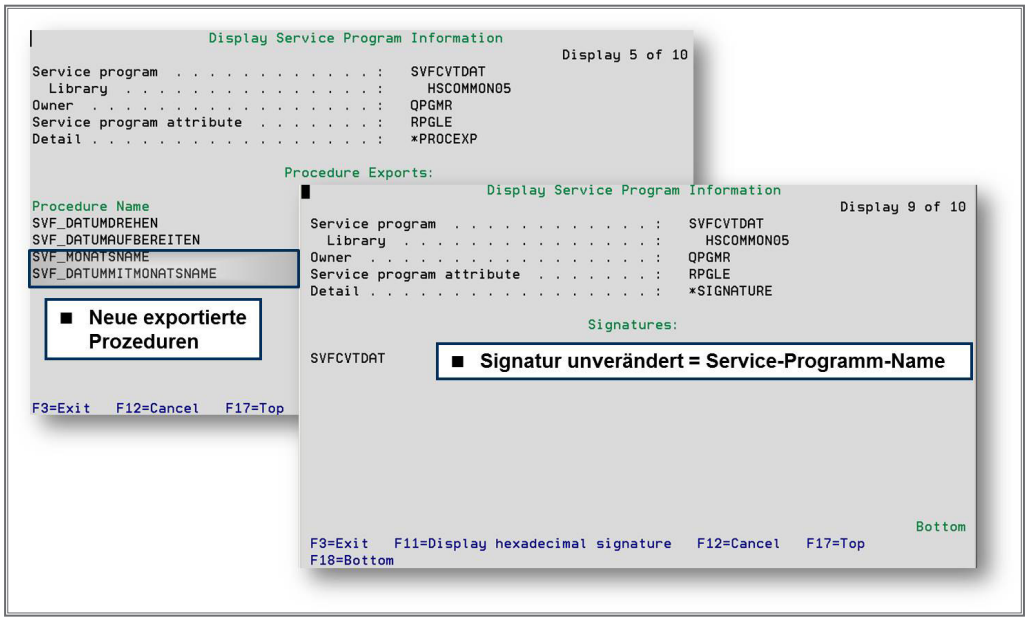
Nachdem die neuen Prozeduren in den Source-Code integriert und das Modul SVFCVTDAT neu erstellt wurde, wird die Binderquelle angepasst. Die beiden neuen Prozeduren SVF_MonatsName und SVF_DatumMitMonatsName werden am Ende des Exportblocks hinzugefügt. Die Signatur bleibt unverändert.

```
STRPGMEXP SIGNATURE('SVFCVTDAT')
EXPORT    SYMBOL(SVF_DatumDrehen)
EXPORT    SYMBOL(SVF_DatumAufbereiten)
EXPORT    SYMBOL(SVF_MonatsName)
EXPORT    SYMBOL(SVF_DatumMitMonatsName)
ENDPGMEXP
```

Hinzufügen von exportierten Prozeduren in Binderquellen mit fixer Signatur

Zeigt man sich die Serviceprogramm-Informationen nach dem erneuten Binden mit dem Befehl DSPSRVPGMSRC erneut an, so stellt man fest, dass zwar die neuen exportierten Prozeduren im Serviceprogramm integriert sind, die Signatur jedoch unverändert ist und weiterhin dem Serviceprogrammnamen entspricht.

3.1.8
Seite 16



DSPSRVPGM – Neue exportierte Prozeduren – Signatur unverändert

Im Programm SVFCALL2 wird eine der neuen exportierten Prozeduren benötigt. Deshalb wird der Source-Code entsprechend geändert. Das Modul und das Programm werden im Anschluss neu erstellt. Nach der Neuerstellung kann das Programm SVFCALL2 problemlos aufgerufen werden.

Da in Programm SVFCALL1 keine der neuen Prozeduren benötigt wird, bleiben sowohl der Source-Code als auch das Programm selbst unverändert. Da auch die Signatur im Serviceprogramm SVFCVTDAT unverändert bleibt und bereits im Programm SVFCALL1 eingebunden ist, kann auch dieses Programm nach dem Hinzufügen von neuen exportierten Prozeduren in Serviceprogramm SVFCVTDAT ohne erneutes Binden problemlos aufgerufen werden.

3.1.8.5 Beispiel: Serviceprogramm mit mehreren Signaturen

Die manuelle Vorgabe der Serviceprogramm-Signaturen ist relativ einfach zu handhaben, hat jedoch auch einen kleinen Nachteil. In seltenen Fällten könnte es zu doppelten Signaturnamen kommen. Damit kann das Serviceprogramm nicht mehr eindeutig lokalisiert werden, was zumindest zu unerwarteten Ergebnissen führen kann.

Um diesem Problem aus dem Weg zu gehen, würde manch einer die automatische Generierung der Signatur bevorzugen, da diese unter anderem auf der Basis der aus dem Serviceprogramm exportierten Prozeduren ermittelt wird. Der Nachteil hierbei ist, dass sich die Signatur ändert, wenn sich die Anzahl der exportierten Prozeduren ändert.

Um Problemen mit Signaturverletzungen aus dem Weg zu gehen und um das erneute Binden der Programme und Serviceprogramme zu verhindern, muss man dafür sorgen, dass sowohl die aktuelle Signatur als auch die vorherigen Signaturen im Serviceprogramm gespeichert werden.

Wir gehen von der gleichen Ausgangssituation wie in den vorangegangenen Beispielen aus:

- Das Serviceprogramm SVGCVTDAT mit den exportierten Prozeduren SVG_DatumDrehen und SVG_DatumAufbereiten soll um die Prozeduren SVG_Monatsname und SVG_DatumMitMonatsname erweitert werden.
- Das Serviceprogramm bzw. Prozeduren aus dem Serviceprogramm werden in den Programmen SVGCALL1 und SVGCALL2 aufgerufen.
- Die neuen Prozeduren werden nur in SVGCALL2 benötigt.

Der Source-Code für alle Objekte entspricht dem Source-Code im Beispiel: Neue Prozeduren in Serviceprogramm SVACVTDAT. Um eine klare Unterscheidung zu erhalten wurden die Teildateien, Prozeduren und Objekte umbenannt (statt des Präfixes SVA wurde überall das Präfix SVG verwendet.

Im Gegensatz zum vorherigen Beispiel soll die Signatur des Serviceprogramms nicht fix vorgegeben, sondern ermittelt werden. Die Binderquelle für das ursprüngliche Serviceprogramm sieht damit wie folgt aus:

```
STRPGMEXP
EXPORT SYMBOL(SVG_DatumDrehen)
EXPORT SYMBOL(SVG_DatumAufbereiten)
ENDPGMEXP
```

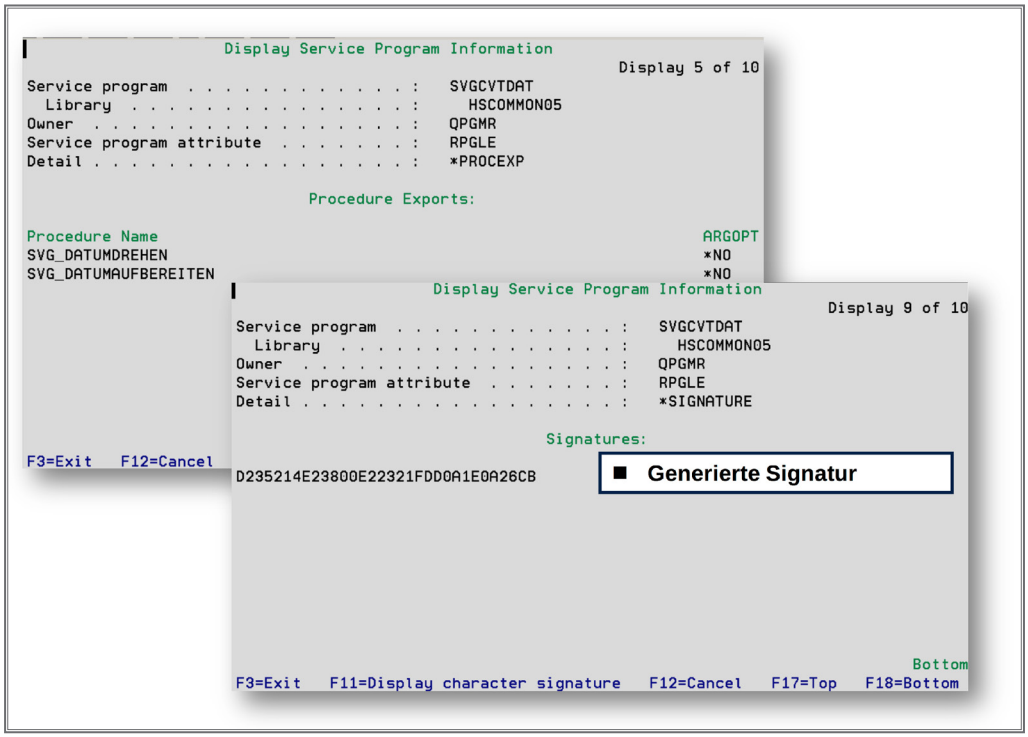
Binder-Quelle mit automatischer Signatur-Ermittlung

Ergänzung 33/2015

3.1.8
Seite 18

Das Serviceprogramm sowie die rufenden Programme werden genau wie im vorherigen Beispiel zweistufig erstellt. Der einzige Unterschied ist, dass die Signatur automatisch ermittelt und in die rufenden Programme integriert wurde.

Die folgende Abbildung zeigt einen Auszug aus dem Befehl DSPSRVPGM mit den exportierten Prozeduren und der automatisch generierten Signatur.



Serviceprogramm mit automatisch generierter Signatur

Um nach dem Hinzufügen des Source-Codes für die neuen Prozeduren und der Erstellung des Serviceprogramms die neu enthaltenen Prozeduren von außerhalb des Serviceprogramms aufrufen zu können, muss die Binderquelle modifiziert werden.

Damit die aktuelle und vorherige Signatur in das Serviceprogramm integriert werden kann, muss der zuvor aktuelle Exportblock als vorhergehender Exportblock in der Quelle verbleiben. Für die aktuelle Signatur muss ein neuer Exportblock an erster Stelle in die Quelle integriert werden, in dem alle vier exportierten Prozeduren angegeben sind. Die neuen Prozeduren müssen dabei immer ans Ende der Exportliste angefügt werden.

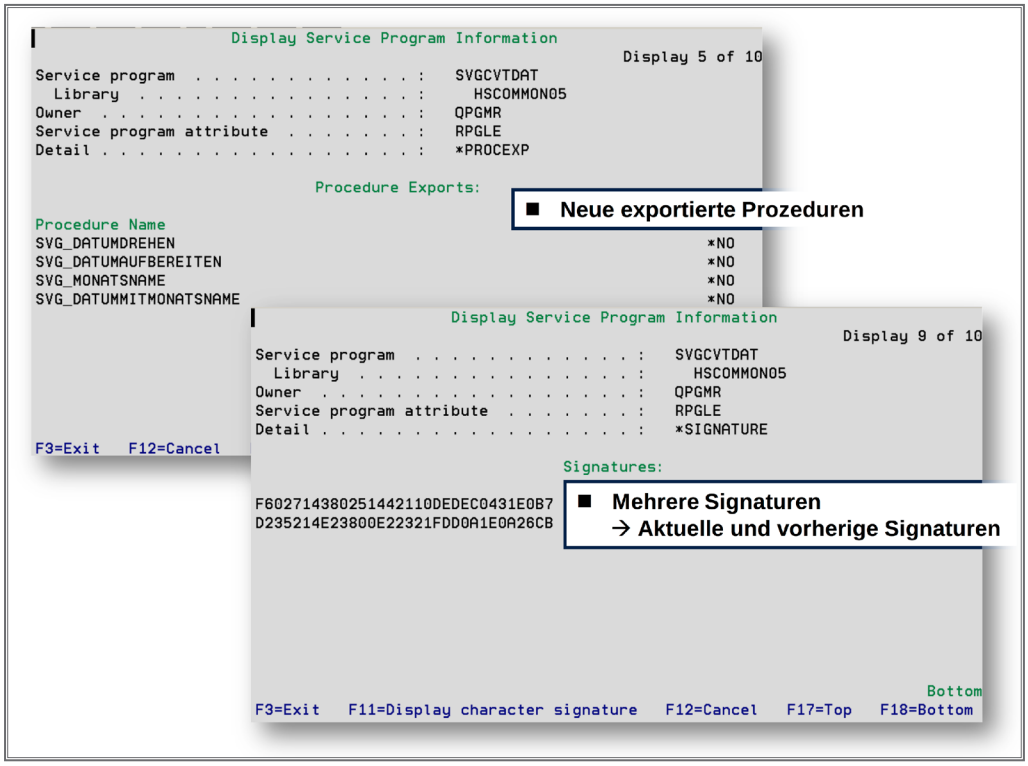
Das folgende Beispiel zeigt die modifizierte Binderquelle.

```
STRPGMEXP
EXPORT   SYMBOL(SVG_DatumDrehen)
EXPORT   SYMBOL(SVG_DatumAufbereiten)
EXPORT   SYMBOL(SVG_MonatsName)
EXPORT   SYMBOL(SVG_DatumMitMonatsName)
ENDPGMEXP
STRPGMEXP PGMLVL(*PRV)
EXPORT   SYMBOL(SVG_DatumDrehen)
EXPORT   SYMBOL(SVG_DatumAufbereiten)
ENDPGMEXP
```

Binderquelle mit aktuellem und vorherigem Exportblock

Erstellt man jetzt das Serviceprogramm mit dieser modifizierten Binderquelle, so werden sowohl die aktuelle als auch die vorherige Signatur in das Serviceprogramm übernommen.

Das folgende Beispiel zeigt das Serviceprogramm SVGCVTDAT nach dem erneuten Binden mit der modifizierten Binderquelle.



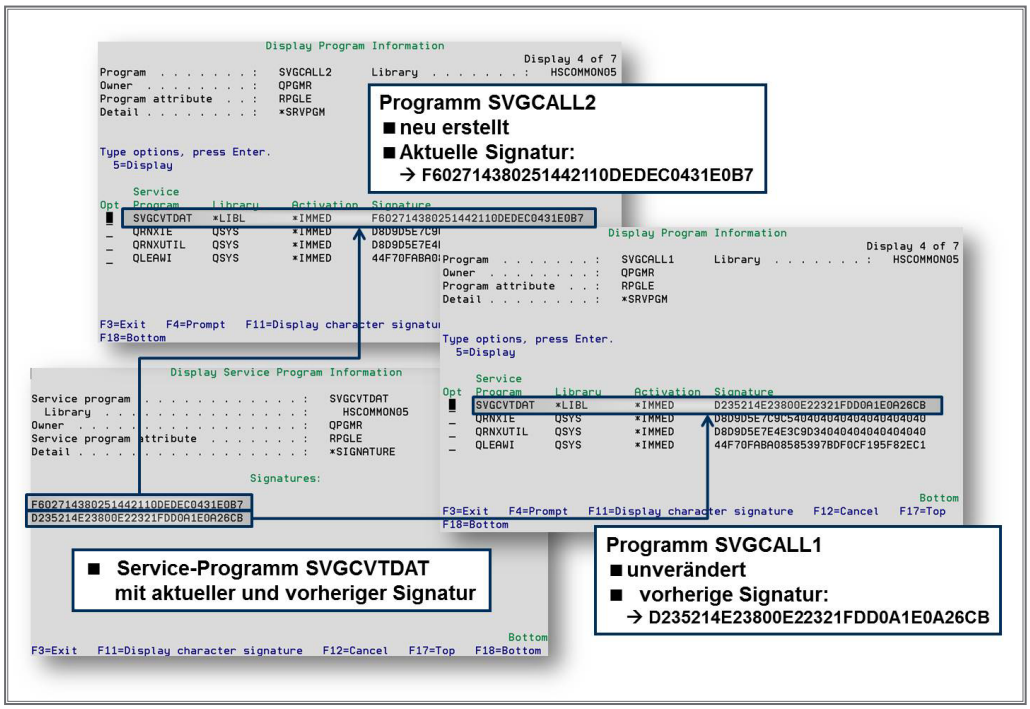
Serviceprogramm mit mehreren generierten Signaturen

3.1.8
Seite 20

Programme, in denen die neuen Prozeduren aufgerufen werden, müssen neu erstellt werden. Bei der Erstellung wird die aktuelle Signatur eingebunden. Damit können diese Programme problemlos aufgerufen werden.

In allen anderen Programmen, die eine Prozedur aus dem Serviceprogramm aufrufen und die nicht neu gebunden wurden, ist eine der vorherigen Signaturen eingebunden. Da diese Signaturen weiterhin im Serviceprogramm integriert sind, kann das Serviceprogramm bei der Aktivierung lokalisiert und das Programm problemlos ausgeführt werden.

Das folgende Beispiel zeigt nochmals die Signaturen des Serviceprogramms SVGCVTDAT sowie die eingebundenen Signaturen in den Programmen SVGCALL1 und SVGCALL2. In Programm SVGCALL2 wird eine neue Prozedur aufgerufen, deshalb wurde das Programm neu erstellt und die aktuelle Signatur eingebunden. Das Programm SVGCALL1 benötigt keine dieser neuen Prozeduren, wurde deshalb nicht angefasst und beinhaltet damit noch die alte Signatur.



Programme mit verschiedenen Signaturen des gleichen Serviceprogramms

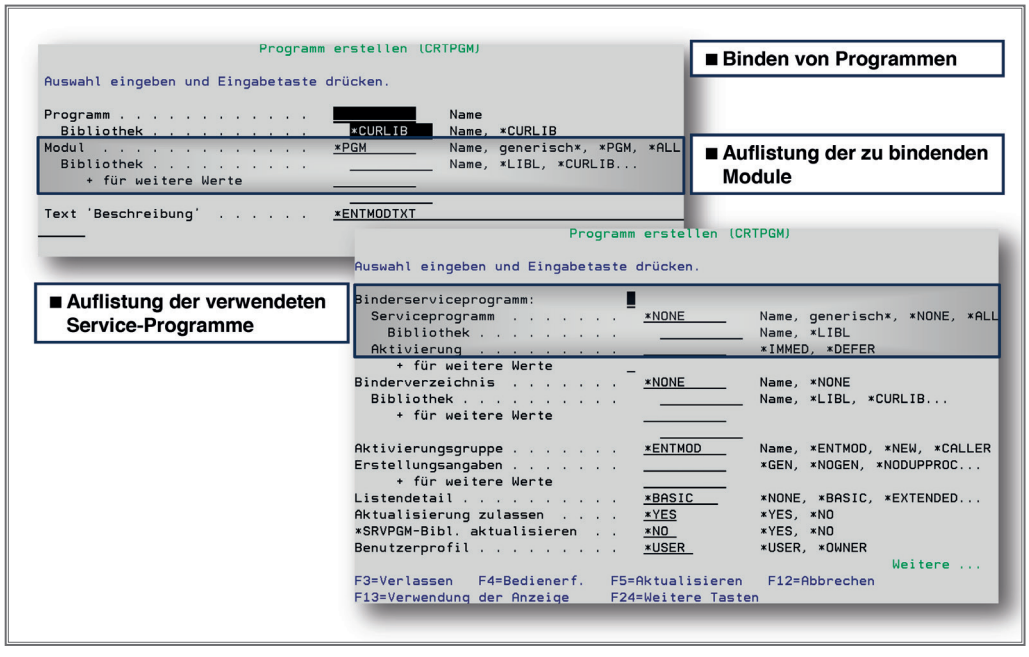
3.1.9 Binderverzeichnis

In den vorangegangenen Beispielen haben wir uns bislang darauf beschränkt, ein einziges Modul oder Serviceprogramm mit diversen exportierten Prozeduren in Programmobjekte einzubinden.

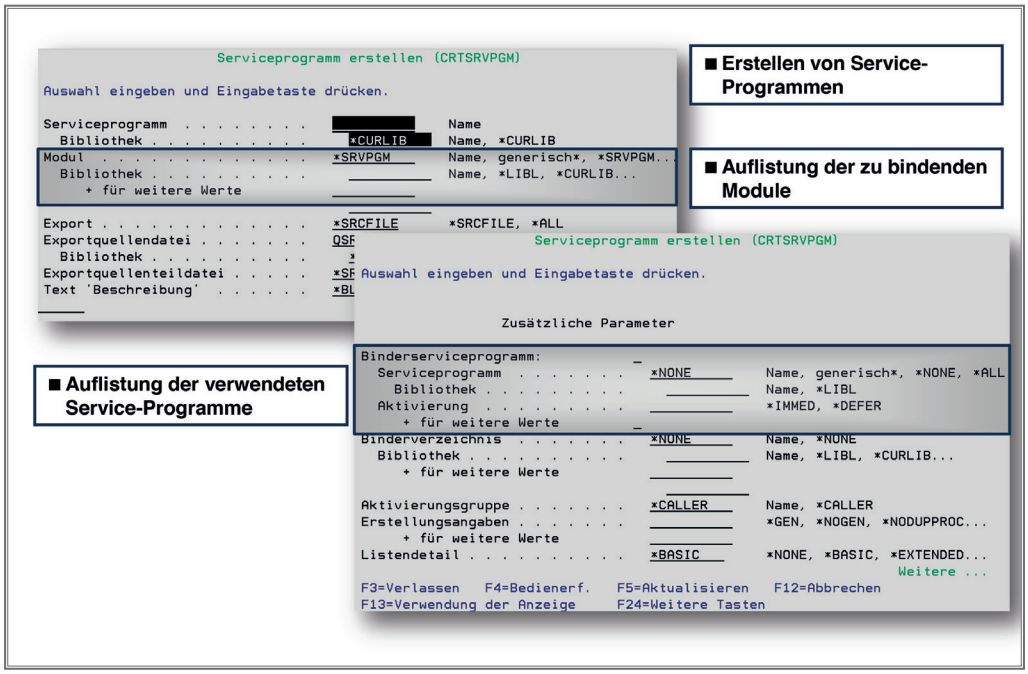
In einer hochmodularen ILE-Umgebung wird es jedoch eine Vielzahl von Serviceprogrammen geben, in denen die Prozeduren abhängig von ihrer Funktionalität gruppiert wurden. In den Programmen werden unter Umständen viele Prozeduren aus fast ebenso vielen verschiedenen Serviceprogrammen aufgerufen. Außerdem kann es vorkommen, dass aus Prozeduren in einem Serviceprogramm weitere Prozeduren aufgerufen werden, die sich in verschiedenen anderen Serviceprogrammen befinden.

Im Binderschritt, also beim Ausführen der Befehle CRTPGM (Programm erstellen) oder CRTSRVPGM (Serviceprogramm erstellen), aber auch bei den Befehlen UPDPGM (Programm aktualisieren) oder UPDSRVPGM (Serviceprogramm aktualisieren) müssen die verwendeten Serviceprogramme und Module aufgelistet werden. Dies ist erforderlich, damit entweder das Modul kopiert oder die Signatur des Serviceprogramms in das Objekt übernommen werden kann.

Das folgende Beispiel zeigt die Optionen MODULE (Modul) und BNDSRVPGM (Binderserviceprogramm) in den Befehlen CRTPGM und CRTSRVPGM.



3.1.9
Seite 2



Optionen MODUL und BNDSRVPGM in den Erstellungsbefehlen CRTPGM und CRTSRVPGM

Die (manuelle) Auflistung der gebundenen Module oder Serviceprogramme kann sich dann zu einem mühsamen Geschäft entwickeln. Um diese manuellen Auflistungen zu vermeiden, wurden Binderverzeichnisse eingeführt.

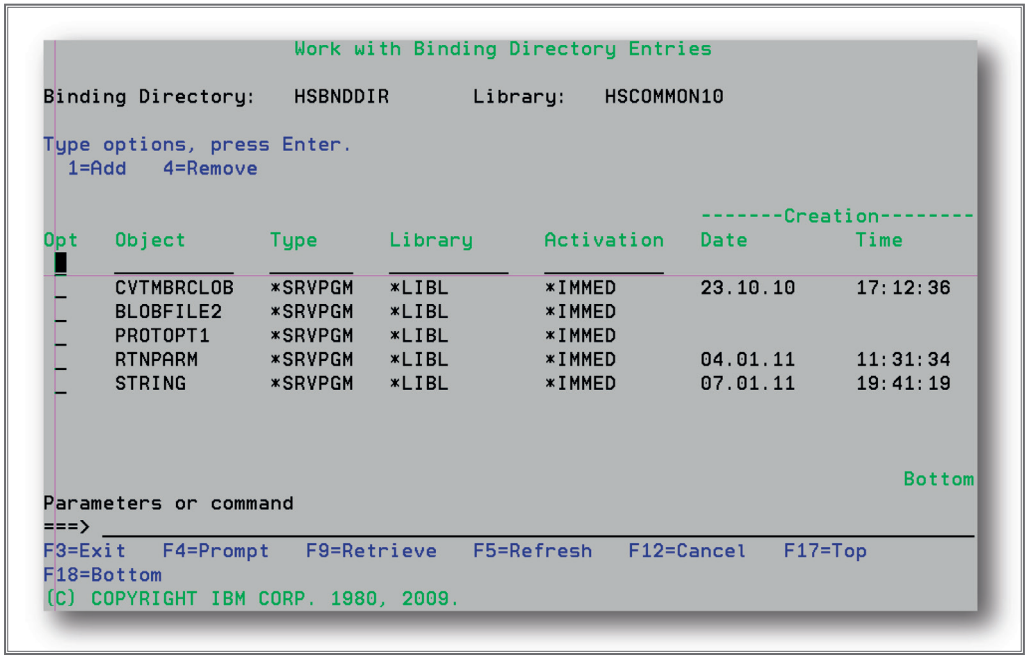
Binderverzeichnisse sind spezielle Objekte der Objektart *BNDDIR, die nur während des Binderschritts benötigt werden. In einem Binderverzeichnis werden Module oder Serviceprogramme aufgelistet. Diese Binderverzeichnisse können in den Erstellungsbefehlen – zum Beispiel CRTPGM, UPDSRVPGM, CRTBNDRPG – über die Binderschritte initiiert werden.

Während des Binderschritts werden die angegebenen Binderverzeichnisse nach Modulen und/oder Serviceprogrammen durchsucht, aus denen Prozeduren aufgerufen werden. Wird ein Modul mit der entsprechenden Prozedur in einem der angegebenen Binderverzeichnisse gefunden, so wird es in das (Service-) Programmobjekt kopiert. Wird ein Serviceprogramm gefunden, das die aufgerufene Prozedur beinhaltet, so wird die Signatur dieses Serviceprogramms in das zu bindende Serviceprogrammobjekt übernommen.

Zur Erstellung und Verarbeitung von Binderverzeichnissen stehen mehrere CL-Befehle zur Verfügung:

- CRTBNDDIR — Binderverzeichnis erstellen
- DLTBNDDIR — Binderverzeichnis löschen.
- DSPBNDDIR — Binderverzeichnis anzeigen.
  Bei dem Befehl DSPBNDDIR werden die Binder-Verzeichniseinträge für das ausgewählte Binderverzeichnis aufgelistet.
- WRKBNDDIR — Mit Binderverzeichnissen arbeiten.
  Mit dem Befehl WRKBNDDIR werden zunächst alle Binderverzeichnisse angezeigt, die der Auswahl entsprechen.
  Über diverse Optionen kann die weitere Bearbeitung des Binderverzeichnisses sowie der Binder-Verzeichniseinträge erfolgen.
- ADDBNDDIRE — Binder-Verzeichniseintrag hinzufügen.
- RMVBNDDIRE — Binder-Verzeichniseintrag entfernen.
- WRKBNDDIRE — Mit Binder-Verzeichniseinträgen arbeiten.

Das folgende Beispiel zeigt die über den Befehl WRKBNDDIRE (Mit Binderverzeichnis arbeiten) erfassten Einträge in Binderverzeichnis HSBNDDIR in Bibliothek HSCOMMON10. Neben den eingetragenen Modulen und Serviceprogrammen sind auch der Aktivierungszeitpunkt und der letzte Änderungszeitpunkt ersichtlich.



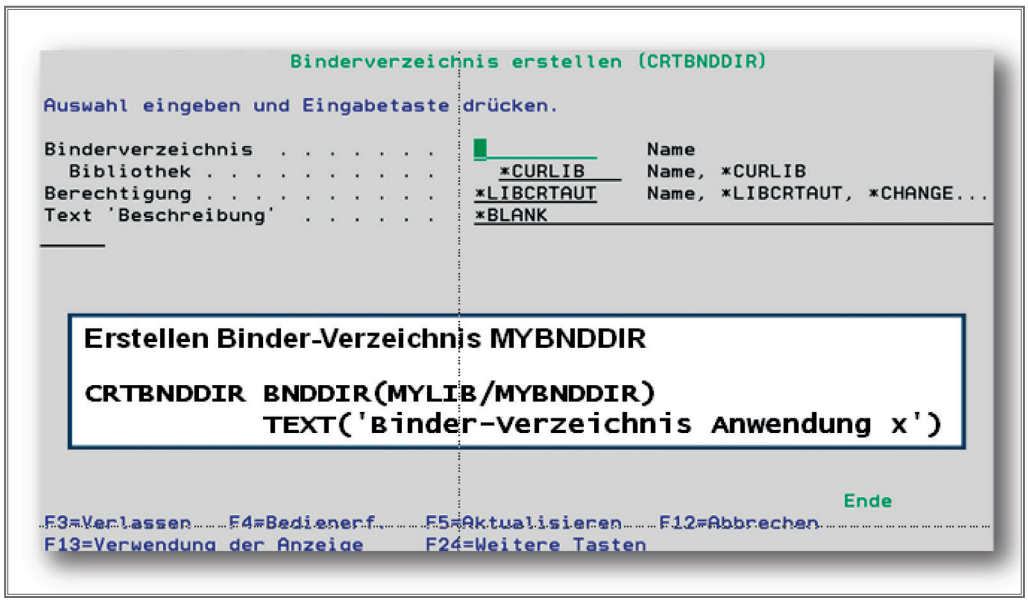
Befehl WRKBNDDIRE – Mit Binder-Verzeichniseinträgen arbeiten

3.1.9
Seite 4

3.1.9.1 Binderverzeichnis erstellen

Ein Binderverzeichnis wird mit dem Befehl CRTBNDDIR (Binderverzeichnis erstellen) erstellt. Dazu müssen lediglich der Name und die Bibliothek des zu erstellenden Binderverzeichnisses angegeben werden. Des Weiteren können noch die Berechtigung sowie eine Beschreibung hinterlegt werden.

Das folgende Beispiel zeigt den Befehl CRTBNDDIR (Binderverzeichnis erstellen).



Befehl CRTBNDDIR (Binderverzeichnis erstellen)

3.1.9.2 Binder-Verzeichniseinträge erstellen

Nach der Erstellung des Binderverzeichnisses müssen die gewünschten Einträge mit Hilfe des Befehls ADDBNDDIRE (Binder-Verzeichniseinträge hinzufügen) erfasst werden. Hierfür sind die folgenden Informationen erforderlich:

- Binderverzeichnis — Qualifiziertes Binderverzeichnis. Für die Bibliothek kann wahlweise ein Bibliotheksname angegeben oder der Sonderwert *LIBL verwendet werden. Wird für ein Serviceprogramm eine Bibliothek hart angegeben, so wird zur Laufzeit das Serviceprogramm aus der angegebenen Bibliothek verarbeitet. Dabei spielt es keine Rolle, ob die Bibliothek zur Laufzeit in der Bibliotheksliste enthalten ist. Das qualifiziert angegebene Serviceprogramm wird auch dann verwendet, wenn in der Bibliotheksliste in einer vorgelagerten Bibliothek ein Serviceprogramm gleichen Namens gefunden wird.
- Objektspezifikationen
  - Objekt — Objekt(e), die in dem Binderverzeichnis hinterlegt werden sollen. In einem ADDBNDDIRE-Befehl können bis zu 50 unabhängige Objekte aufgelistet werden. Außer dem Objektnamen kann auch eine generische Angabe erfolgen, zum Beispiel MYSRV*, wobei im Anschluss alle mit MYSRV beginnenden Serviceprogramme oder Module in das Binderverzeichnis übernommen werden. Mit Hilfe des zulässigen Sonderwerts *ALL können auch alle Serviceprogramme oder Module in das Binderverzeichnis übernommen werden. Für die Bibliothek kann wahlweise die Bibliothek angegeben werden oder der Sonderwert *LIBL.
  - Objektart — Modul (*MODULE) oder Serviceprogramm (*SRVPGM).

3.1.9
Seite 6

– Art der Aktivierung — Seit Release 6.1 kann angegeben werden, zu welchem Zeitpunkt die Aktivierung des Serviceprogramms erfolgen soll.

*IMMED — Die Aktivierung erfolgt direkt mit derjenigen des rufenden Programms (Unterlassungswert).

*DEFERED — Die Aktivierung erfolgt erst mit Aufruf der ersten Prozedur aus dem Serviceprogramm.

• Positionsangaben

– Objektposition — Position, an der die zuvor angegebenen Objekte in das Binderverzeichnis eingefügt werden sollen.

*LAST — Objekte werden am Ende der Binder-Verzeichniseinträge eingetragen.

*FIRST — Objekte werden am Anfang der Binder-Verzeichniseinträge eingetragen.

*AFTER — Objekte werden nach dem angegebenen Objekteintrag in das Binderverzeichnis eingetragen.

*BEFORE — Objekte werden vor dem angegebenen Objekteintrag in das Binderverzeichnis eingetragen.

*REPLACE — Objekte ersetzten das angegebene Objekt.

– Objekt — Bei den Objektpositionen *AFTER, *BEFORE und *REPLACE muss das Referenzobjekt angegeben werden.

– Objektart — Objektart (*MODULE oder *SRVPGM) des Referenzobjekts.

Das folgende Beispiel zeigt den Befehl ADDBNDDIRE (Binder-Verzeichniseinträge hinzufügen):

```
                Binderverz.eintrag hinzufügen (ADDBNDDIRE)

Auswahl eingeben und Eingabetaste drücken.

Binderverzeichnis  . . . . . .   _________    Name
  Bibliothek . . . . . . . . .     *LIBL      Name, *LIBL, *CURLIB...
Objektspezifikationen:           _
  Objekt . . . . . . . . . . .   _________    Name, generisch*, *ALL
    Bibliothek . . . . . . . .     *LIBL      Name, *LIBL
  Objektart  . . . . . . . . .   *SRVPGM      *SRVPGM, *MODULE
  Aktivierung  . . . . . . . .   *IMMED       *IMMED, *DEFER
    + für weitere Werte          _
Positionsangaben:
  Objektposition . . . . . . .   *LAST        *LAST, *FIRST, *AFTER...
  Objekt . . . . . . . . . . .   __________    Name
    Bibliothek . . . . . . . .     _________   Name, *LIBL
  Objektart  . . . . . . . . .   _______       *SRVPGM, *MODULE


                                                                  Ende
F3=Verlassen   F4=Bedienerf.   F5=Aktualisieren   F12=Abbrechen
F13=Verwendung der Anzeige     F24=Weitere Tasten
```

Binder-Verzeichniseintrag hinzufügen

Anmerkung:

Beim Eintrag eines Moduls oder Serviceprogramms in ein Binderverzeichnis wird weder geprüft, ob das Objekt in der Bibliotheksliste vorhanden ist noch ob es überhaupt existiert.

Module und Serviceprogramme können auch gelöscht werden, ohne dass dies einen Einfluss auf die Binder-Verzeichniseinträge hat.

Binder-Verzeichniseinträge werden erst während des Binderschritts verarbeitet. Ist ein eingetragenes Serviceprogramm oder Modul nicht vorhanden, erfolgt keine Fehlermeldung. Der Eintrag wird einfach ignoriert.

in Binderfehler tritt erst dann auf, wenn eine aufgerufene Prozedur in keinem der existierenden eingetragenen Serviceprogramme oder Module gefunden wird.

3.1.9.3 Regeln zur Verwendung von Binderverzeichnissen

Beim Binden von Programmen und Serviceprogrammen können jeweils mehrere Binderverzeichnisse angegeben werden. In diesen Binderverzeichnissen können wiederum mehrere Serviceprogramme und Module hinterlegt sein.

Wird beim Binderschritt jedoch festgestellt, dass die gleiche exportierte Prozedur mehrfach vorhanden ist, beispielsweise weil sowohl das Modul als auch das Serviceprogramm in einem der angegebenen Binderverzeichnisse eingetragen sind, oder weil das gleiche Serviceprogramm in mehreren Binderverzeichnissen hinterlegt ist, so kann das Programm bzw. Serviceprogramm per Default nicht generiert werden.

Es ist möglich, in den Binderbefehlen über die Erstellungsangabe *DUPPROC (= Doppelte Prozeduren) exportierte Prozeduren gleichen Namens zuzulassen. Man sollte diese Option allerdings nur im äußersten Notfall in Betracht ziehen. Stattdessen sollte man sich die folgenden Regeln zu eigen machen:

- Module und Serviceprogramme sollten nie im gleichen Binderverzeichnis angegeben werden.
- Ein Serviceprogramm sollte immer nur in einem einzigen Binderverzeichnis hinterlegt sein.
- Man sollte nur wenige Binderverzeichnisse verwenden, in denen jedoch alle Serviceprogramme eingetragen sind, zum Beispiel
  - ein allgemeines Binderverzeichnis mit allen allgemein einsetzbaren Serviceprogrammen (wie String-Verarbeitung, Datumsroutinen)
  - ein Binderverzeichnis pro Anwendung (zum Beispiel je ein Binderverzeichnis mit allen speziellen Serviceprogrammen für die Buchhaltung oder die Materialwirtschaft)
  - ein Binderverzeichnis pro Bibliothek mit allen Serviceprogrammen in dieser Bibliothek.

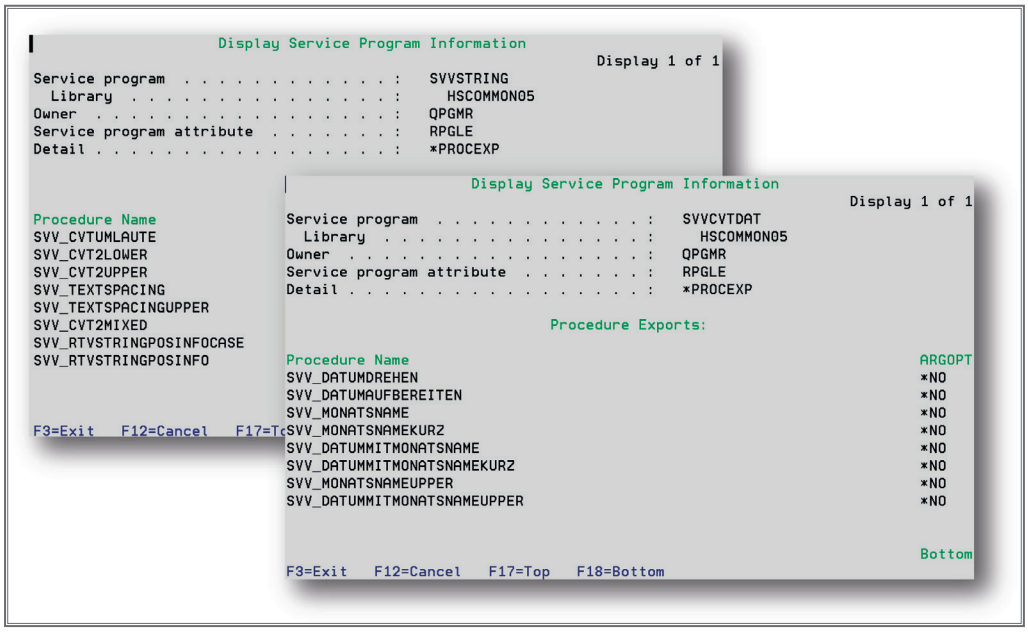
3.1.9.4 Zweistufige Kompilierung mit Binderverzeichnissen

Programme und Serviceprogramme können in einer zweistufigen Kompilierung erstellt werden, was heißt, dass zunächst die Module erstellt werden. Im Anschluss daran – während des Binderschritts – werden die Module entweder zu Programmen oder Serviceprogrammen gebunden. Dabei müssen die Module angegeben werden, die zu einem Programm oder Serviceprogramm gebunden werden sollen. Werden in den Prozeduren, die sich innerhalb der Module befinden, andere Prozeduren aus verschiedenen Serviceprogrammen aufgerufen, so müssen auch diese Serviceprogramme beim Binden angegeben werden.

Anstatt die gebundenen Objekte aufzulisten, können Module und Serviceprogramme in Binderverzeichnisse eingetragen werden. Im Binderschritt werden dann nur einige wenige Binderverzeichnisse aufgelistet.

In unserer Umgebung sind aktuell zwei Serviceprogramme vorhanden, SVVCVTDAT (Datumsfunktionen) und SVVSTRING (String-Funktionen). In beiden Serviceprogrammen sind diverse exportierte Prozeduren hinterlegt.

In der folgenden Abbildung ist zu sehen, wie über den Befehl DSPSRVPGM die exportierten Prozeduren in den Serviceprogrammen SVVSTRING und SVVCVTDAT in Bibliothek HSCOMMON05 angezeigt werden können.



Exportierte Prozeduren aus den Serviceprogrammen SVVSTRING und SVVCVTDAT

3.1.9
Seite 10

In Serviceprogramm SVVSTRING sind nur Prozeduren vorhanden, die entweder keine anderen Prozeduren oder nur Prozeduren innerhalb des gleichen Serviceprogramms aufrufen. Da diese Serviceprogramme in Zukunft noch um weitere Prozeduren erweitert werden und außerdem noch weitere Serviceprogramme geplant sind, aus denen Prozeduren in den beiden Serviceprogrammen aufgerufen werden, wurde beschlossen, das Binderverzeichnis ILEBNDDIR mit dem folgenden Befehl anzulegen:

```
CRTBNDDIR BNDDIR(YOUROBJLIB/ILEBNDDIR)
          TEXT('ILE Binder-Verzeichnis')
```

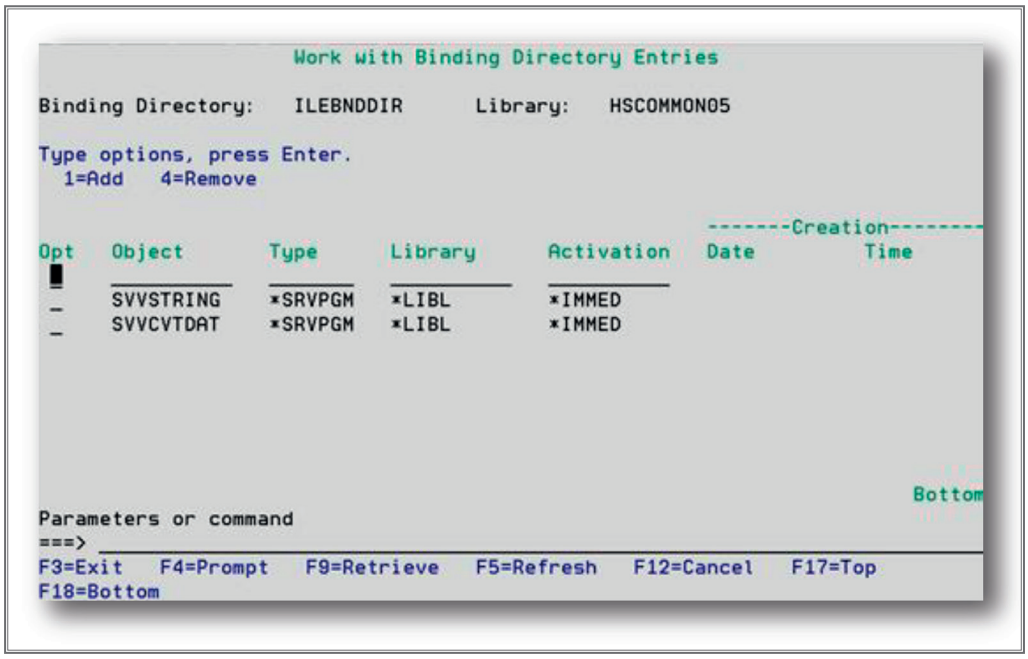
Erstellen von Binderverzeichnis ILEBNDDIR

Diesem Binderverzeichnis werden die beiden Serviceprogramme SVVCVTDAT und SVVSTRING mit dem folgenden Befehl hinzugefügt:

```
ADDBNDDIRE BNDDIR(YOUROBJLIB/ILEBNDDIR)
           OBJ((YOUROBJLIB/SVVSTRING *SRVPGM *IMMED)
               (YOUROBJLIB/SVVCVTDAT *SRVPGM *IMMED))
```

Hinzufügen von Serviceprogrammen zu Binderverzeichns ILEBNDDIR

In der folgenden Abbildung wird das Binderverzeichnis, das beide Einträge enthält, über den Befehl WRKBNDDIRE (Mit Binderverzeichnissen arbeiten) angezeigt. Da beide Serviceprogramme bislang noch nicht erstellt wurden, und zum Zeitpunkt des Eintrags nicht auf ihre Existenz geprüft wird, werden an dieser Stelle kein Erstellungsdatum und keine Erstellungszeit angezeigt.



Binder-Verzeichniseinträge in Binderverzeichnis ILEBNDDIR

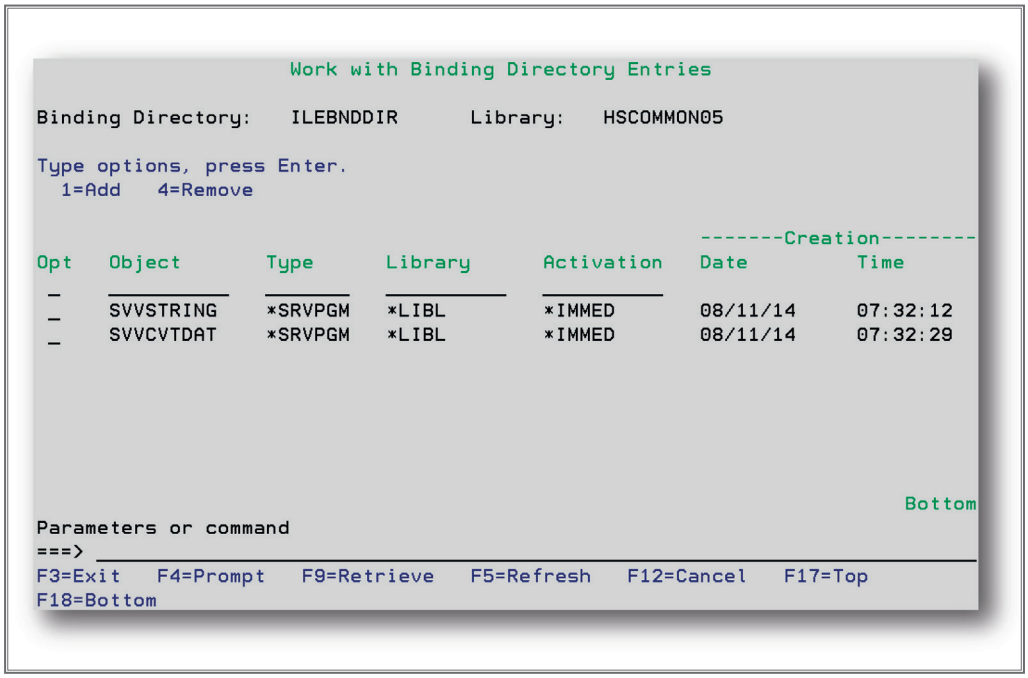
Im Anschluss daran werden zunächst die Module erstellt, die dann in Service-Programme gebunden werden. Die dafür erforderlichen Befehle werden im folgenden Beispiel gezeigt. Da die Prozeduren aus dem Serviceprogramm SVVSTRING aus Prozeduren in Serviceprogramm SVVCVTDAT aufgerufen werden, muss das Serviceprogramm SVVSTRING zuerst erstellt werden.

```
CRTRPGMOD MODULE(YOURMODLIB/SVVSTRING)
          SRCFILE(YOURSRCLIB/QRPGLESRC)
          SRCMBR(SVVSTRING)

CRTRPGMOD MODULE(YOURMODLIB/SVVCVTDAT)
          SRCFILE(YOURSRCLIB/QRPGLESRC)
          SRCMBR(SVVCVTDAT)

CRTSRVPGM SRVPGM(YOUROBJLIB/SVVSTRING)
          MODULE(*SRVPGM)
          SRCFILE(YOURMODLIB/QSRVSRC)
          SRCMBR(SVVSTRING)
          BNDDIR(YOUROBJLIB/ILEBNDDIR)

CRTSRVPGM SRVPGM(YOUROBJLIB/SVVCVTDAT)
          MODULE(*SRVPGM)
          SRCFILE(YOURMODLIB/QSRVSRC)
          SRCMBR(SVVCVTDAT)
          BNDDIR(YOUROBJLIB/ILEBNDDIR)
```

Zweistufige Compilierung von Serviceprogrammen mit Binderverzeichnis ILEBNDDIR

3.1.9
Seite 12

Prüft man nach der Erstellung der Serviceprogramme mit dem Befehl WRKBNDDIRE (Mit Binder-Verzeichniseinträgen arbeiten) die Binder-Verzeichniseinträge erneut, werden auch die Erstellungsdaten angezeigt, wie die folgende Abbildung belegt.



Binder-Verzeichniseinträge in ILEBNDDIR nach Erstellung der Serviceprogramme

Prozeduren aus den Serviceprogrammen SVVC VTDAT und SVVSTRING werden in den Programmen SVVCALL1 und SVVCALL2 aufgerufen. Diese Programme können ebenfalls in einer zweistufigen Kompilierung unter Angabe des Binderverzeichnisses über die folgenden Befehle erstellt werden.

```
CRTRPGMOD MODULE(YOURMODLIB/SVVCALL1)
          SRCFILE(YOURSRCLIB/QRPGLESRC)
          SRCMBR(SVVSTRING)

CRTRPGMOD MODULE(YOURMODLIB/SVVCALL2)
          SRCFILE(YOURSRCLIB/QRPGLESRC)
          SRCMBR(SVVCVTDAT)

CRTPGM PGM(YOUROBJLIB/SVVCALL1)
       MODULE(*PGM)
       BNDDIR(YOUROBJLIB/ILEBNDDIR)

CRTPGM PGM(YOUROBJLIB/SVVCALL2)
       MODULE(*PGM)
       BNDDIR(YOUROBJLIB/ILEBNDDIR)
```

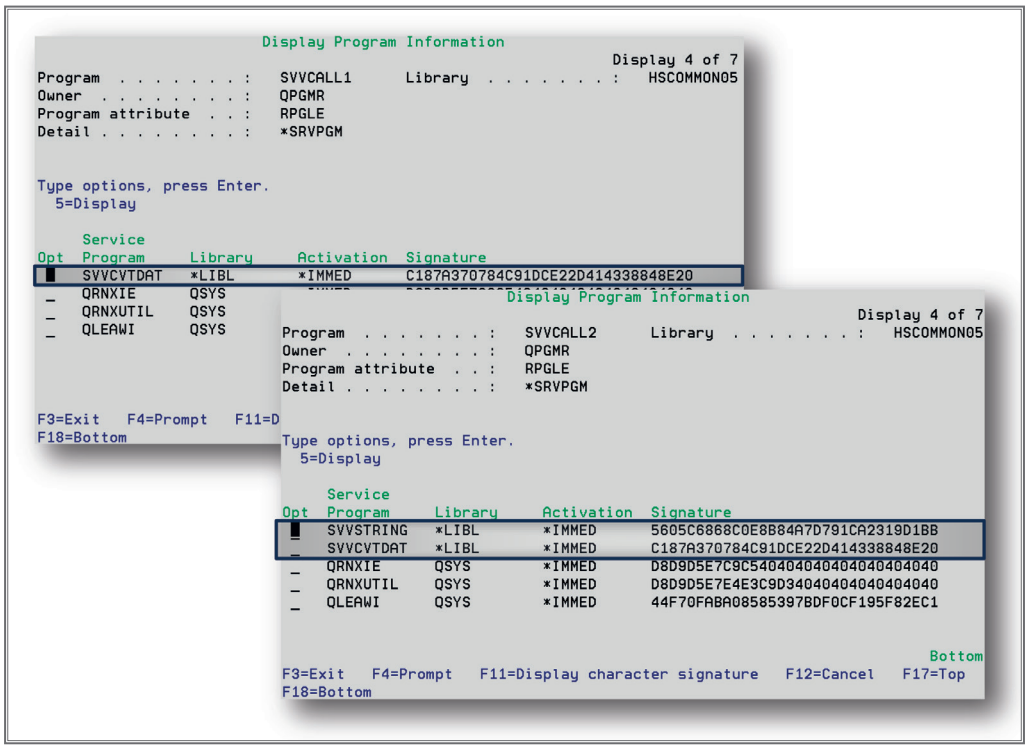
Zweistufige Kompilierung von Programmen mit Binderverzeichnis

Anmerkung:

Nach dem Binderschritt sind die Module nicht länger erforderlich und können/sollten gelöscht werden.

3.1.9
Seite 14

Zeigt man sich nach dem Erstellen der Programme mit dem Befehl DSPPGM (Programm anzeigen) die gebundenen Serviceprogramme an, so stellt man fest, dass in Programm SVVCALL1 nur das Serviceprogramm SVVCVTDAT gebunden ist, während in Programm SVVCALL2 beide Serviceprogramme (SVVCVTDAT und SVVSTRING) gebunden sind, wie die folgende Abbildung belegt.



Gebundene Serviceprogramme in den Programmen SVVCALL1 und SVVCALL2

3.1.9.5 Einstufige Kompilierung mit Binderverzeichnissen

Während Serviceprogramme immer zweistufig kompiliert werden müssen, können Programme mit dem Befehl CRTBNDxxx (XXX Binderprogramm erstellen) direkt in einem Schritt erstellt werden.

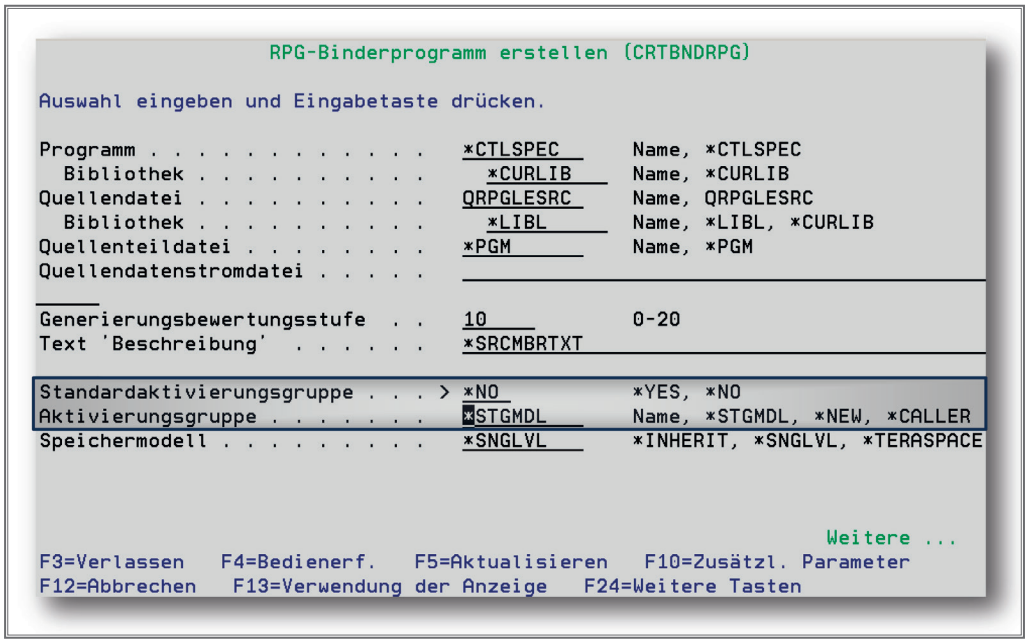
Dieses Vorgehen war kein Problem, solange in dem Programm nur ein Modul gebunden war und keine Prozeduren aus Serviceprogrammen aufgerufen wurden. In beiden Fällen müssen die zu bindenden Objekte im Binderschritt angegeben werden, was eine zweistufige Kompilierung erfordert, … zumindest solange man ohne Binderverzeichnisse arbeitet.

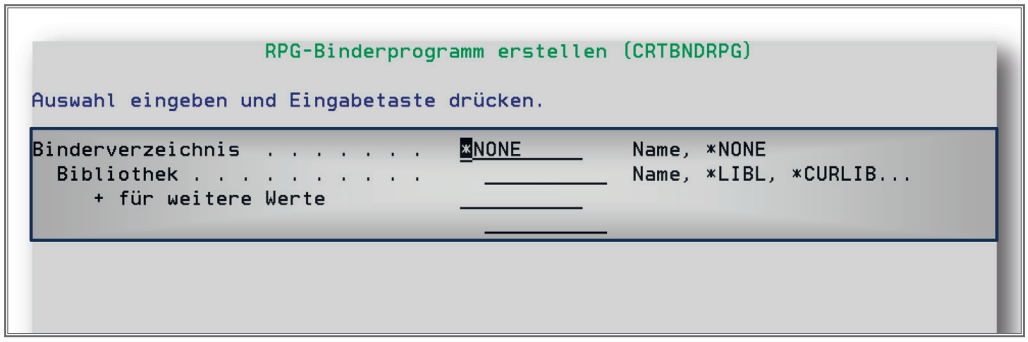
3.1.9
Seite 16

3.1.9.6 Befehl CRTBNDxxx mit Binderverzeichnis

Sofern man im Befehl CRTBNDxxx (xxx Binderprogramm erstellen) den Wert für die Standard-Aktivierungsgruppe von *YES auf *NO ändert und eine Aktivierungsgruppe angibt (oder für die Aktivierungsgruppe auch nur den Unterlassungswert akzeptiert), können auch bei der direkten Erstellung von Binderprogrammen ein oder mehrere Binderverzeichnisse angegeben werden.

Die folgende Abbildung zeigt den Befehl CRTBNDRPG (RPG-Binderprogramm erstellen), nachdem der Wert für die Standard-Aktivierungsgruppe auf *NO geändert wurde.





Befehl CRTBNDRPG (RPG-Binderprogramm erstellen) mit Binderverzeichnis

Anmerkung:

Sofern Module und Serviceprogramme in (unterschiedlichen) Binderverzeichnissen hinterlegt werden, können Programme auch dann in einem Schritt erstellt werden, wenn mehrere Module gebunden oder Prozeduren aus diversen Serviceprogrammen aufgerufen werden.

Sofern Programme embedded SQL beinhalten, wird es etwas problematischer, da die Optionen DFTACTGRP (Default-Aktivierungsgruppe), ACTGRP (Aktivierungsgruppe) und BNDDIR (Binderverzeichnis) nicht direkt im CRTSQLRPGI-Befehl angegeben werden können.

Beim Erstellen von Objekten mit embedded SQL wird zunächst die Originalquelle durch den SQL-Precompiler überarbeitet. Dabei werden die SQL-Statements auskommentiert und durch API (Application Programming Interface)-Aufrufe ersetzt. Die Objekte werden im Anschluss durch den Aufruf der Systembefehle CRTBNDxxx oder CRTxxxMOD – je nachdem ob ein Programm oder Modul erzeugt werden soll – aus der modifizierten Quelle erstellt.

Wie bereits zuvor erwähnt, beinhalten die CRTSQLxxxI-Befehle nicht alle Optionen, die in den Befehlen CRTBNDxxx und CRTxxxMOD gesetzt werden können. Um Compile-Optionen für die Erstellungsbefehle setzen zu können, wurde bereits mit Release V5R3M0 die Option COMPILEOPT (Compiler-Optionen) in die Befehle CRTSQLxxxI integriert. Fehlende Optionen können über die COMPILEOPT-Option in Form eines Strings mit einer Länge von bis zu 5.000 Zeichen hinterlegt werden. Mehrere Optionen können durch Leerzeichen getrennt aufgelistet werden. Es erfolgt allerdings keine Prüfung, ob die angegebenen Optionen gültig sind. Ungültige Optionen werden ignoriert.

3.1.9
Seite 18

Die folgende Abbildung zeigt die Compiler-Optionen im Befehl CRTSQLRPGI sowie den kompletten Erstellungsbefehl für das Programm MYPGM. Die Option DFTACTGRP wird auf *NO gesetzt und das Programm mit Aktivierungsgruppe YOURPGM erstellt. Zum Auffinden der zu bindenden Module und Serviceprogramme wird das Binderverzeichnis YOURBNDDIR verwendet.

```
                    SQL ILE RPG-Objekt erstellen (CRTSQLRPGI)

 Auswahl eingeben und Eingabetaste drücken.

 Compileroptionen . . . . . > 'DFTACTGRP(*NO) ACTGRP(YOURPGM) BNDDIR(YOURB
NDDIR)'
                                                                        ...

 CRTSQLRPGI   OBJ(YOUROBJLIB/YOURPGM)
              SRCFILE(YOURSRCLIB/QRPGLESRC)
              SRCMBR(YOURPGM)
              OBJTYPE(*PGM)
              COMPILEOPT('DFTACTGRP(*NO) ACTGRP(YOURPGM)
                          BNDDIR(YOURBNDDIR)')

                                                                    Ende
 F3=Verlassen    F4=Bedienerf.    F5=Aktualisieren    F12=Abbrechen
 F13=Verwendung der Anzeige       F24=Weitere Tasten
```

Erstellen eines Programms mit embedded SQL und Binderverzeichnissen

3.1.9.6.1 H-Bestimmungen für ILE-Optionen in RPG-Quellen

Zwar können mit diesen Voraussetzungen Programme in einem Schritt erstellt werden, die Optionen Standard-Aktivierungsgruppe, Aktivierungsgruppe und Binderverzeichnisse müssen jedoch bei jeder Kompilierung (manuell) gesetzt werden.

Sofern mit nur einem einzigen Binderverzeichnis oder immer mit den gleichen Binderverzeichnissen gearbeitet und immer die gleiche Aktivierungsgruppe angegeben wird, können ggf. die Unterlassungswerte im Systembefehl geändert werden. Spätestens dann, wenn man flexibler werden will oder muss, also unterschiedliche Binderverzeichnisse oder Aktivierungsgruppen benötigt, ist man genauso weit wie vor der Änderung des Systembefehls, das heißt, ein manueller Eingriff ist erforderlich.

Anstatt sich bei jeder Umwandlung an die entsprechenden Umwandlungsoptionen erinnern zu müssen, können die folgenden Optionen auch direkt in den H-Bestimmungen des Source-Codes hinterlegt werden.

Hierfür werden die folgenden Schlüsselworte zur Verfügung gestellt:

Default-Aktivierungsgruppe – Schlüsselwort DFTACTGRP(*YES | *NO)
Der Aufruf von externen ILE-Prozeduren und Funktionen ist nur bei Verwendung der Option DFTACTGRP(*NO) zulässig.

Werden Programme mit DFTACTGRP(*YES) erstellt, verhalten sie sich wie OPM (Original Program Model)-Programme (etwa RPGIII). Die Verwendung von DFTACTGRP(*YES) ist dann sinnvoll, wenn für eine alte RPGII/RPGIII-Anwendung ein Programm nach dem anderen nach RPGIV konvertiert werden soll, um dann die neuen RPGIV-Features – wie Free-Format-Codierung – nutzen zu können. In (modernen) ILE-Anwendungen sollten jedoch weder Programme noch Serviceprogramme in der Default-Aktivierungsgruppe ausgeführt werden.

Die Angabe des Schlüsselworts DFTACTGRP in den H-Bestimmungen ist nur zulässig, wenn das Programm mit dem Befehl CRTBNDRPG (RPG-Binderprogramm erstellen) generiert wird. Bei einer zweistufigen Kompilierung bzw. bei Ausführung des Befehls CRTRPGMOD (RPG-Modul erstellen) führt die Angabe des Schlüsselworts DFTACTGRP in den H-Bestimmungen zu einem Kompilierungsfehler, sofern es nicht durch eine Compiler-Direktive bedingt wurde.

3.1.9
Seite 20

Seit der letzten Erweiterung in Release 7.1 ist die Angabe des Schlüsselworts DFTACTGRP(*NO) für ILE-Programme optional. Voraussetzung dafür ist allerdings, dass eines der Schlüsselworte ACTGRP (Aktivierungsgruppe), BNDDIR (Binderverzeichnis) oder STGMDL (Storage Model) in den H-Bestimmungen hinterlegt wurde. In diesem Fall wird DFTACTGRP(*NO) vom Compiler automatisch gesetzt.

Sofern das Programm nicht in der Default-Aktivierungsgruppe ausgeführt werden soll, ist die Angabe einer Aktivierungsgruppe erforderlich. Im Umkehrschluss bedeutet dies: Das Schlüsselwort ACTGRP kann bei Schlüsselwort DFTACTGRP(*YES) nicht gesetzt werden.

Der Name der Aktivierungsgruppe kann entweder fix vorgegeben werden (= benannte Aktivierungsgruppe) oder über einen der zulässigen Sonderwerte festgelegt werden.

- *STGMDL: Abhängig vom ausgewählten Storage Model (Single Level oder Teraspace) wird das Programm zur Laufzeit entweder in der Aktivierungsgruppe QILE (Single Level) oder QILETS (Teraspace) aktiviert.
- *NEW Bei jedem Aufruf wird das Programm in einer neuen Aktivierungsgruppe aktiviert, die nach Programmende wieder gelöscht wird.
- *CALLER Das Programm wird in der gleichen Aktivierungsgruppe ausgeführt wie das rufende (Service-)Programm.

Die Angabe des Schlüsselworts ACTGRP in den H-Bestimmungen ist nur erlaubt, wenn das Programm mit dem Befehl CRTBNDRPG (RPG-Binderprogramm erstellen) generiert wird.

Binderverzeichnis – Schlüsselwort BNDDIR(‚BINDERVZ1‘ {:'BINDERVZ2'...})

In den H-Bestimmungen können mit Hilfe des Schlüsselworts BNDDIR entweder ein einziges oder mehrere Binderverzeichnisse angegeben werden.

Die einzelnen Binderverzeichnisse werden voneinander durch Doppelpunkte getrennt, in Hochkommata eingebettet und in Großbuchstaben angegeben. Binderverzeichnisse können auch qualifiziert in der Form 'BIBLIOTHÈK/BINDERVERZ' angegeben werden.

Werden Compile-Optionen in den H-Bestimmungen hinterlegt, so überschreiben in der Regel die in den H-Bestimmungen hinterlegten Optionen die Angaben im Kompilierungsbefehl. Eine Ausnahme von dieser Regel bilden jedoch die Binderverzeichnisse, das heißt, die Binder-Verzeichniseinträge im Kompilierungsbefehl werden NICHT überschrieben, sondern hinzugefügt.

Achtung:

Werden in den H-Bestimmungen und im Kompilierungsbefehl unterschiedliche Binderverzeichnisse angegeben, so **werden ALLE angegebenen Binderverzeichnisse** durchsucht, also sowohl die in den H-Bestimmungen als auch die im Kompilierungsbefehl aufgelisteten Binderverzeichnisse.

Das folgende Code-Beispiel zeigt ILE-Compile-Optionen für ein Programm, zunächst im fixen Format und im Anschluss die gleichen Optionen im Free-Format.

Das Programm wird nach Erstellung aufgrund des Schlüsselworts ACTGRP in der Aktivierungsgruppe YOURPGM ausgeführt. Die Angabe des Schlüsselworts DFTACTGRP(*NO) ist optional. Deshalb wurde das Schlüsselwort nur im fixen, nicht jedoch im Free-Format angegeben. Zum Auffinden der zu bindenden Module und Serviceprogramme werden die Binderverzeichnisse YOURBNDD01 und YOURBNDD03, die sich zur Compile-Zeit in der Bibliotheksliste befinden, sowie das Binderverzeichnis YOURBNDD02 in Bibliothek YOUROBJLIB durchsucht.

```
 * Fix-Format Codierung
H DftActGrp(*No) Actgrp('YOURPGM')
H BndDir('YOURBNDD01': 'YOUROBJLIB/YOURBNDD02': 'YOURBNDD03')

  // Free-Format Codierung
  CTL-Opt ActGrp('YOURPGM')
          BndDir('YOURBNDD01': 'YOUROBJLIB/YOURBNDD02': 'YOURBNDD03');
```

H-Bestimmungen mit ILE-Erstellungsoptionen

Werden Programme erstellt, die embedded SQL beinhalten, so können die Erstellungsoptionen wahlweise in den H-Bestimmungen oder über das Schlüsselwort COMPILEOPT in einer SET OPTION-Anweisung hinterlegt werden.

Anmerkung:

Da die Compiler-Optionen (COMPILEOPT) weder im Kompilierungsbefehl CRTSQLRPGI noch in der SET OPTION-Anweisung auf Gültigkeit geprüft werden, ist die Angabe in den H-Bestimmungen dem Statement SET OPTION bzw. den Compiler-Optionen vorzuziehen.

3.1.9.6.1.1 Compiler-Direktiven für ein- oder zweistufige Kompilierung

Die Schlüsselworte DFTACTGRP (Default-Aktivierungsgruppe) und ACTGRP (Aktivierungsgruppe) dürfen nur dann in den H-Bestimmungen hinterlegt sein, wenn das Programm mit dem Befehl CRTBNDRPG erstellt wird.

Versucht man eine Quelle, in der eines der Schlüsselworte ACTGRP oder DFTACTGRP verwendet wurde, zweistufig zu kompilieren, also zunächst das Modul zu erstellen und im Anschluss daran das Programm zu binden, so wird dies nicht gelingen. Bereits beim Erstellen des Moduls tritt ein Kompilierungsfehler auf.

Über Compiler-Direktiven ist es möglich, die Schlüsselworte so zu hinterlegen, dass sie nur dann berücksichtigt werden, wenn die Kompilierung über CRTBNDRPG erfolgt.

Compiler-Direktiven beginnen immer mit einem Slash (/). Im fixen Format muss der Slash auf Position 7 angegeben werden, im Free-Format kann die Compiler-Direktive an jeder beliebigen Stelle zwischen Position 8 und 80 begonnen werden. Pro Zeile kann jedoch immer nur eine einzige Compiler-Direktive angegeben werden. Zu den Compiler-Direktiven gehören unter anderem /COPY und /INCLUDE, aber auch /FREE und /END-FREE.

Neben diesen Compiler-Direktiven gibt es noch die bedingten Compiler-Direktiven. Die Bedingungen werden über eine /IF-Anweisung geprüft. Auf /IF muss entweder DEFINED oder NOT DEFINED folgen. Die eigentliche Bedingung wird im Anschluss in runden Klammern eingebettet angegeben, zum Beispiel /IF DEFINED (*CRTBNDRPG).

Neben den /IF-Direktiven gibt es noch die /ELSEIF und /ELSE-Direktiven sowie die /ENDIF-Direktive. Jede /IF-Direktive muss mit einer /ENDIF-Direktive abgeschlossen werden. Die Angabe von /ELSE bzw. /ELSEIF-Direktiven ist optional.

Code-Zeilen zwischen der /IF-Direktive und der /ENDIF-Direktive werden vom Compiler nur dann berücksichtigt, wenn die in der /IF-Direktive angegebene Bedingung wahr ist.

Um bestimmte Code-Zeilen während der Kompilierung basierend auf dem verwendeten Kompilierungsbefehl zu aktivieren oder zu ignorieren, wurden die folgenden vordefinierten Bedingungen bereitgestellt:

- *CRTBNDRPG — Die Code-Zeilen werden nur berücksichtigt, wenn die Kompilierung mit dem Befehl CRTBNDRPG erfolgt.
- *CRTRPGMOD — Die Code-Zeilen werden nur berücksichtigt, wenn die Kompilierung mit dem Befehl CRTRPGMOD erfolgt.

Das folgende Beispiel zeigt nochmals die H-Bestimmungen aus dem vorangegangenen Beispiel. Dieses Mal sind die Schlüsselworte DFTACTGRP und ACTGRP jedoch bedingt, das heißt, sie werden nur dann berücksichtigt, wenn die Kompilierung mit CRTBNDRPG erfolgt. Wird die Quelle mit CRTMODRPG in ein Modul umgewandelt, werden beide Schlüsselworte ignoriert und das Modul kann ordnungsgemäß erstellt werden. Das Schlüsselwort BNDDIR ist weder im fixen noch im Free-Format bedingt, wird also grundsätzlich berücksichtigt.

Für die Free-Format-Anweisungen ist noch anzumerken, dass in diesem Beispiel die Compiler-Direktiven innerhalb des CTL-OPT-Statements angegeben wurden und nur für einen Teil der Schlüsselworte gelten. Diese Syntax ist zulässig. Alternativ hätten auch mehrere CTL-Opt-Statements, die einzeln bedingt werden, angegeben werden können.

```
 * Fixed Format
 /If Defined (*CRTBNDRPG)
H DftActGrp(*No) Actgrp('YOURPGM')
 /EndIf
H BndDir('YOURBNDD01': 'YOUROBJLIB/YOURBNDD02': 'YOURBNDD03')

  // Free Format
  CTL-Opt BndDir('YOURBNDD01': 'YOUROBJLIB/YOURBNDD02': 'YOURBNDD03')
      /If Defined (*CRTBNDRPG)
          ActGrp('YOURPGM')
     /EndIf
         ;
```

Compiler-Direktiven für ein- oder zweistufige Kompilierung

3.1.9.6.2
CL-Befehl DCLPRCOPT für ILE-Optionen in CL-Quellen

Sofern ILE-CL-Programme erstellt werden müssen, die in einer speziellen Aktivierungsgruppe ausgeführt werden, oder aus denen Prozeduren aufgerufen werden, die in mehreren Service-Programmen hinterlegt sind und die sich wiederum in diversen Binderverzeichnissen befinden, können diese Informationen direkt im Kompilierungsbefehl angegeben werden.

Es besteht aber auch für ILE-CL die Möglichkeit, die Kompilierungsoptionen direkt in der Quelle zu hinterlegen. Hierfür dient der CL-Befehl DCLPRCOPT (Verarbeitungsoptionen deklarieren).

Die Verwendung des Befehls DCLPRCOPT ist nur innerhalb von ILE-CL-Quellen (Teildateiart CLLE) erlaubt. Der Befehl DCLPRCOPT darf in einer Quelle nur einmal angegeben werden. Wurde der Befehl DCLPRCOPT mehrfach codiert, bricht die Kompilierung mit Fehlermeldung CPD0323 ab. Wie alle anderen DECLARE-Befehle, etwa DCL (CL-Variable deklarieren) oder DCLF (Datei deklarieren), muss auch DCLPRCOPT am Anfang des CL-Quellcodes vor allen ausführenden Befehlen angegeben werden. Die Reihenfolge, in der die unterschiedlichen Declare-Befehle zu Beginn des Programms angegeben werden, ist nicht fix vorgegeben, das heißt, die Reihenfolge ist beliebig.

Die folgende Tabelle zeigt den Befehl DCLPRCOPT mit allen Optionen und deren Ausprägungen.

DCLPRCOPT Schlüssel-Wort	Beschreibung	Auswahl	Bemerkung
SUBRSTACK	Subroutine-Stapel-Tiefe	20-9999, **99**	Optional, Positional 1
LOG	Befehle protokollieren	*JOB, *YES, *NO	Optional
ALWRTVSRC	RTVCLSRC erlauben	*YES, *NO	Optional
TEXT	Text 'Beschreibung'	*Alphanumer. Wert*, *SRCMBRTXT, *BLANK	Optional
USRPRF	Benutzer-Profil	*USER, *OWNER	Optional
AUT	Berechtigung	*Name*, *LIBCRTAUT, *CHANGE, *ALL, *USE, *EXCLUDE	Optional
DBGENCKEY	Debugverschlüsselungsschlüssel	*Alphanumer. Wert*, *NONE	Optional
STGMDL	Speicher-Model	*SNGLVL, *TERASPACE	Optional
DFTACTGRP	Standard-Aktivierungsgruppe	*YES, *NO	Optional
ACTGRP	Aktivierungsgruppe	*Name*, *STGMDL, *NEW, *CALLER	Optional
BNDSRVPGM	Binder-Service-Programm	Sonder-Wert: *NONE Auflistung von bis zu 300 Service-Programmen	Optional
	Element 1: Service Programm	*Qualifizierter Objekt-name*	
	Qualifier 1: Service Programm	*Generischer Name, Name*, *ALL	
	Qualifier 2: Bibliothek	*Name*, ***LIBL**	
	Element 2: Aktivierung	***IMMED**, *DEFER	
BNDDIR	Binder-Verzeichnis	Sonder-Wert: *NONE Auflistung von bis zu 300 Binder-Verzeichnissen	Optional
	Qualifier 1: Binder-Verzeichnis	*Name*	
	Qualifier 2: Bibliothek	*Name*, ***LIBL**, *CURLIB, *USRLIBL	

Auswahloptionen bei Befehl DCLPRCOPT

Wurde der Befehl DCLPRCOPT in einer CLLE-Quelle angegeben, überschreiben die angegebenen Werte die in den Kompilierungsbefehlen angegebenen Werte oder die in den Kompilierungsbefehlen hinterlegten Default-Werte. Die Ausnahme ist auch hier die Option Binderverzeichnis. Aufgelistete Binderverzeichnisse werden nicht überschrieben, sondern hinzugefügt.

Das folgende Beispiel zeigt den Anfang des Source-Codes eines CLLE-Programms. Über den Befehl DCLPRCOPT wird zunächst die Default-Aktivierungsgruppe auf *NO gesetzt und die Aktivierungsgruppe YOURCLGPM verwendet. Das Binder-Serviceprogramm YOURSRVP01 wird gebunden und sofort aktiviert (Unterlassungswert für Aktivierung ist *IMMED). Das Binder-Serviceprogramm YOURSRVP02 in Bibliothek YOUROBJLIB wird ebenfalls gebunden, seine Aktivierung jedoch bis zum ersten Aufruf einer Prozedur aus diesem Serviceprogramm verzögert (Aktivierung = *DEFER). Die Binderverzeichnisse YOURBNDD01 und YOURBNDD03 in der Bibliotheksliste sowie das Binderverzeichnis YOURBNDD02 in Bibliothek YOUROBJLIB werden während des Binderschritts durchsucht.

```
PGM          PARM(&Parm1 &Parm2 &Parm3)

DCL          VAR(&Parm1) TYPE(*CHAR) LEN(10)

/* Weitere Variablen */

DCLPRCOPT    DFTACTGRP(*NO) +
              ACTGRP(YOURCLPGM) +
              BNDSRVPGM((YOURSRVP01) (YOUROBJLIB/YOURSRVP02 *DEFER)) +
              BNDDIR(YOURBNDD01 YOUROBJLIB/YOURBNDD02 YOURBNDD03)
```

Setzen von ILE-Optionen in CLLE-Quellen über den Befehl DCLRPCOPT

3.1.9
Seite 26

3.1.10 Ausführung von Programmen und Serviceprogrammen

Bevor ein Programm oder eine Prozedur in einem Serviceprogramm ausgeführt werden kann, muss das Programm oder Serviceprogramm zunächst in Laufbereitschaft versetzt werden. Diesen Prozess nennt man Aktivierung.

3.1.10.1 Aktivierung von Programmen und Serviceprogrammen

Die Aktivierung von Programmen und Serviceprogrammen kann in die beiden folgenden Bereiche aufgeteilt werden:

Reservierung und Initialisierung des statischen Speicherbereichs für ein Programm
Das Programm wird in den Speicher geladen.

Für die im Programm definierten Variablen wird Speicher reserviert und entsprechend der Vorgaben im Source-Code initialisiert.

Die für den Record Level Access definierten physischen und logischen Dateien bzw. SQL-Tabellen oder Indices werden geladen. Der RPG-Zyklus bewirkt, dass die in RPG in den F-Bestimmungen definierten Dateien direkt geöffnet werden, sofern das Schlüsselwort USROPN nicht angegeben wurde. Wurde das Schlüsselwort USROPN angegeben, erfolgt die Öffnung der Datei erst bei Ausführung des OpCodes OPEN.

Diese Aktionen erfolgen sowohl für die OPM (Original Program Model)-Programme, etwa RPGIII, klassisches RPG oder Cobol-Programme, als auch für ILE-Programme und Serviceprogramme.

Aktivierung von Serviceprogrammen
Sofern aus ILE-Programmen Prozeduren aufgerufen werden, die in Serviceprogrammen hinterlegt sind, werden beim Aufruf des Programms (Aktivierungszeitpunkt) per Default auch diese Serviceprogramme in Laufbereitschaft versetzt.

Da die aufzurufenden Prozeduren durch die Serviceprogramme aktiviert werden, sich also bei Aufruf in Laufbereitschaft befinden, ist deren Verarbeitung annähernd so schnell wie die Verarbeitung von Subroutinen oder internen Prozeduren.

Sei Release 6.1 kann die Aktivierung von Serviceprogrammen so lange hinausgeschoben werden, bis die erste Prozedur aus diesem Serviceprogramm aufgerufen wird.

Ergänzung 35/2015

3.1.10.2 Deaktivierung von Programmen und Serviceprogrammen

Bei OPM-Programmen kann es sich um alte CLP, RPGIII oder Cobol-Programme handeln, die mit den Befehlen CRTCLPGM, CRTRPGPGM oder CRTCBLPGM erstellt wurden.

Aber auch CLLE oder RPGIV-Programme werden als OPM-Programme erstellt (obwohl der ILE-Compiler verwendet wird), sofern die Option Standard-Aktivierungsgruppe im Kompilierungsbefehl mit *YES angegeben wurde. Bei DFTACTGRP=*YES handelt es sich um den Unterlassungswert, der in den Kompilierungsbefehlen CRTBNDxxx hinterlegt ist.

ILE-Programme werden über den Kompilierungsbefehl CRTBNDxxx nur dann erstellt, wenn die Standard-Aktivierungsgruppe DFTACTGRP auf *NO gesetzt und entweder eine benannte Aktivierungsgruppe oder einer der Sonderwerte *NEW oder *CALLER für die Aktivierungsgruppe angegeben wurde.

Programme, die über eine zweistufige Kompilierung erstellt werden, werden grundsätzlich als ILE-Programme implementiert. Bei der zweistufigen Kompilierung werden zunächst ein oder mehrere Module mit dem Befehl CRTxxxMOD (xxx-Modul erstellen) erzeugt. Im Anschluss werden diese Module mit Hilfe des Befehls CRTPGM (Programm erstellen) zu einem Programm gebunden.

Achtung:

Bei Programmen, die mit dem Befehl CRTBNDXXX und der Option DFTACTGRP=*YES erstellt wurden, handelt es sich um OPM-Programme und nicht um ILE-Programme.

Beim Aufruf von OPM-Programmen werden nur die aufgerufenen Programme aktiviert. Beim Aufruf von ILE-Programmen werden zusätzlich noch die Serviceprogramme aktiviert, aus denen Prozeduren verwendet werden. In beiden Fällen wird das Programm in Laufbereitschaft versetzt.

Beim Beenden werden OPM- und ILE-Programme jedoch unterschiedlich behandelt:

- OPM-Programme werden nach dem Beenden wieder komplett aus dem Speicher entfernt (sofern mit *LR or STOP RUN beendet).
- ILE-Programme bleiben so lange im Speicher resistent, bis die Aktivierungsgruppe in der sie aktiviert wurden, beendet wird.

3.1.10.2.1 Deaktivierung von OPM-Programmen

Ein OPM-Programm wird per Default nach dessen Beendigung komplett aus dem Speicher entfernt.

Beim erneuten Aufruf eines OPM-Programms innerhalb des gleichen Jobs muss eine erneute Aktivierung erfolgen. Da die Aktivierung mit einigem Overhead verbunden ist, kann sich das insbesondere beim wiederholten Aufruf des gleichen Programms negativ auf die Performance auswirken.

Um diesen Aktivierungsaufwand zu minimieren, ist es auch möglich, OPM-Programme inklusive der aktuellen Variablenwerte und aktiven Datei-Pointer bei Programmende im Speicher zu belassen und einzufrieren. In RPG kann dies durch das Beenden des Programms via Operations-Code RETURN anstatt Beendigung durch den Last Record Indicator (*INLR) erreicht werden. Der Programmierer ist im Falle eines erneuten Aufrufs dafür verantwortlich, dass die Variablen korrekt initialisiert und die Datei-Pointer richtig positioniert werden.

Mit Hilfe des CL-Befehls RCLRSC (Ressourcen zurückfordern) kann der für den Job reservierte Speicherbereich freigegeben werden. Dabei werden die mit RETURN beendeten und noch im Speicher befindlichen OPM-Programme beendet und komplett aus dem Speicher entfernt.

Es können immer nur alle Ressourcen zurückgefordert werden, das heißt, es ist nicht möglich, einzelne Programme gezielt zu beenden und nur den Speicher, den dieses Programm belegt, freizugeben.

Achtung:

ILE-Programme und Serviceprogramme werden durch den Befehl RCLRSC nicht aus dem Speicher entfernt. RCLRSC bewirkt bei ILE-Programmen lediglich, dass die geöffneten Dateien geschlossen werden.

3.1.10.2.2 Deaktivierung von ILE-Programmen

In echten ILE-Umgebungen werden aus den Programmen Prozeduren aufgerufen, die sich in unterschiedlichen Serviceprogrammen befinden können. Da alle diese Serviceprogramme ebenfalls aktiviert werden müssen, um die ILE-Programme komplett in Laufbereitschaft zu versetzen, ist die ILE-Aktivierung insbesondere in hochmodularen Anwendungen um einiges aufwendiger als die OPM-Aktivierung.

Um den Aktivierungsaufwand insbesondere beim wiederholten Aufruf von ILE-Programmen so gering wie möglich zu halten, wurden Aktivierungsgruppen eingeführt. ILE-Programme werden in Aktivierungsgruppen aktiviert, ausgeführt und so lange im Speicher gehalten, bis die entsprechende Aktivierungsgruppe beendet wird.

Die Aktivierungsgruppe, in der ein ILE-Programm oder Serviceprogramm ausgeführt wird, wird beim Binden des Programms- oder Serviceprogramms angegeben.

Solange die Aktivierungsgruppe nicht explizit mit Hilfe des CL-Befehls RCLACTGRP (Aktivierungsgruppe zurückfordern) gelöscht wird, bleibt das Programm mit allen in der gleichen Aktivierungsgruppe aktivierten Serviceprogrammen im Speicher erhalten.

Dies gilt auch für Programme, die in RPG mit dem Last-Record-Indicator (*INLR = *ON) oder in Cobol mit STOP RUN. beendet werden. Beim Verlassen des Programms werden lediglich die Dateien geschlossen. Der reservierte Speicher bleibt erhalten und wird beim erneuten Aufruf lediglich initialisiert. Der Programmierer muss sich an dieser Stelle weder um die Initialisierung der Variablen noch um die aktuelle Position der Datei-Pointer kümmern.

Wird das ILE-Programm allerdings mit RETURN beendet, werden genau wie bei den mit RETURN beendeten OPM-Programmen weder Dateien geschlossen noch Variablen beim erneuten Aufruf initialisiert.

Werden ILE-Programme in der Default-Aktivierungsgruppe ausgeführt, zum Beispiel durch die Erstellung mit Aktivierungsgruppe *CALLER und durch den Aufruf aus einem OPM-Programm, so kann der für diese ILE-Programme reservierte Speicher nur durch das Beenden des Jobs wieder freigegeben werden.

Die Default-Aktivierungsgruppe kann mit dem Befehl RCLACTGRP nicht beendet werden, und der Befehl RCLRSC entfernt ILE-Programme nicht aus dem Speicher.

Achtung:

ILE-Programme können nur durch das Schließen der Aktivierungsgruppe, in der sie ausgeführt wurden, aus dem Speicher entfernt werden, und zwar mit Hilfe des CL-Befehls RCLACTGRP.

Wird ein ILE-Programm in der Standard-Aktivierungsgruppe ausgeführt, kann der von diesem ILE-Programm belegte Speicher nur durch das Beenden des Jobs freigegeben werden.

3.1.10.3 Was sind Aktivierungsgruppen?

Der Begriff Aktivierungsgruppe wurde in den vorangegangenen Absätzen bereits mehrfach verwendet. Aber was sind eigentlich Aktivierungsgruppen?

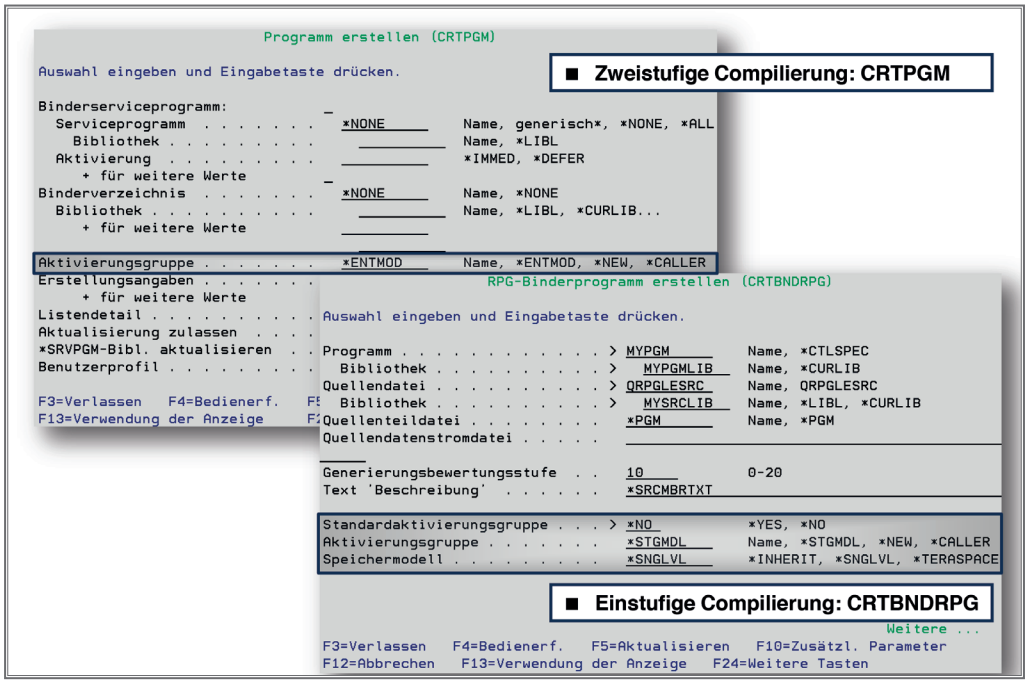
Aktivierungsgruppen sind Sub-Strukturen von Jobs. Aktivierungsgruppen beinhalten alles, was zur Ausführung von einem oder mehreren Programmen notwendig ist. Dazu gehören unter anderem der Speicher für die statischen Variablen, der dynamische Speicherbereich und die offenen Datenpfade (ODPs).

Aktivierungsgruppen wurden erst mit den ILE-Konzepten eingeführt. Das Konzept der OPM-Programme, die innerhalb eines Jobs ausgeführt werden, ist jedoch wesentlich älter. Um eine Co-Existenz von OPM- und ILE-Programmen zu ermöglichen, werden alle OPM-Programme in der Default-Aktivierungsgruppe ausgeführt, die beim Job-Start automatisch gestartet und bei Job-Ende wieder gelöscht wird.

Die Aktivierungsgruppe, in der ein ILE-Programm ausgeführt wird, wird beim Erstellen des Programms entweder im Befehl CRTBNDxxx (xxx-Binderprogramm erstellen) oder im Befehl CRTPGM (Programm erstellen) angegeben. Sofern die Programme einstufig, das heißt über den Befehl CRTBNDRPG kompiliert werden, kann die Aktivierungsgruppe auch in den H-Bestimmungen mit Hilfe der Schlüsselworte DFTACTGRP und ACTGRP im Source-Code hinterlegt werden.

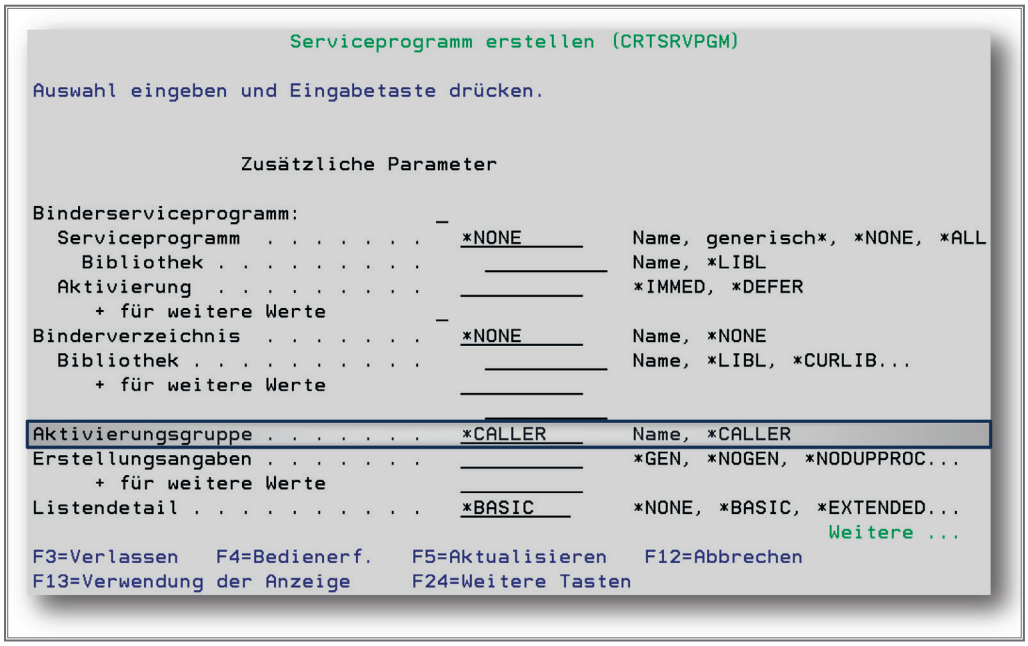
3.1.10
Seite 6

Die folgende Abbildung zeigt die beiden Erstellungsbefehle CRTPGM (Programm erstellen) und CRTBNDRPG (RPG-Binderprogramm erstellen) mit den Optionen für die Aktivierungsgruppe.



Aktivierungsgruppe bei den Erstellungsbefehlen CRTPGM und CRTBNDRPG

Bei der Erstellung von Serviceprogrammen muss die Aktivierungsgruppe ebenfalls im Befehl CRTSRVPGM (Serviceprogramm erstellen) angegeben werden. Im Befehl CRTSRVPGM kann für die Aktivierungsgruppe jedoch nur der Sonderwert *CALLER oder eine benannte Aktivierungsgruppe angegeben werden. Bei Serviceprogrammen ist weder die Option Default-Aktivierungsgruppe noch die Aktivierungsgruppe *NEW zulässig.



Aktivierungsgruppe beim Binden eines Serviceprogramms

Die Aktivierungsgruppe kann auch über die Änderungsbefehle UPDPGM (Update-Programm) und UPDSRVPGM (Update-Serviceprogramm) gesetzt bzw. geändert werden.

Mehrere unabhängige Programme und Serviceprogramme können der gleichen Aktivierungsgruppe zugewiesen werden. So können beispielsweise alle zu einer Anwendung gehörenden Programme und Serviceprogramme in der gleichen Aktivierungsgruppe ausgeführt werden.

Jeder Job beinhaltet mindestens zwei Aktivierungsgruppen, die sogenannten Standard/Default-Aktivierungsgruppen, die automatisch bei Job-Start angelegt werden. In diesen Aktivierungsgruppen werden die System-OPM-Programme und die individuellen OPM-Programme aktiviert.

Wo bzw. in welcher Aktivierungsgruppe Programme und Serviceprogramme aktiviert werden, hängt von dem Aktivierungsgruppenattribut ab, das bei Erstellung des Programms oder Serviceprogramms angegeben wurde. Die folgenden Attribute können hinterlegt werden:

Standard-Aktivierungsgruppe *YES
Standard-Aktivierungsgruppe *YES kann nur mit dem Befehl CRTBNDxxx (Binderprogramm erstellen) angegeben werden. Die Ausführung von Programmen in der Default-Aktivierungsgruppe war in erster Linie dazu gedacht, den Umstieg von RPGIII nach RPGIV zu erleichtern.

Bei Programmen, die mit Standard-Aktivierungsgruppe *YES erstellt wurden, handelt es sich jedoch nicht um ILE-Programme, sondern weiterhin um OPM-Programme. Dadurch ist der Aufruf von Prozeduren und Funktionen, die in Serviceprogrammen hinterlegt sind, bei Programmen, die in der Default-Aktivierungsgruppe ausgeführt werden, nicht zulässig.

Benutzerbenannte Aktivierungsgruppe
Der Name der Aktivierungsgruppe wird vom Programmierer individuell festgelegt.

Individuelle Aktivierungsgruppen können sowohl für Programme als auch für Serviceprogramme angegeben werden.

Die gleiche Aktivierungsgruppe kann für mehrere unabhängige Programme und Serviceprogramme verwendet werden.

Systembenannte Aktivierungsgruppe

- *NEW — Die Aktivierung erfolgt bei jedem Aufruf des Programms.
  Bei Programmende wird die Aktivierungsgruppe automatisch gelöscht.
- *CALLER — Das Programm oder Serviceprogramm wird in der gleichen Aktivierungsgruppe ausgeführt, in der auch die rufende Prozedur ausgeführt wird.
  Werden Programme mit Aktivierungsgruppe *CALLER erstellt, kann dies zur Folge haben, dass ILE-Programme und davon abhängige Serviceprogramme in der Default-Aktivierungsgruppe ausgeführt werden. Dieser Fall tritt ein, wenn ILE-Programme, die mit Aktivierungsgruppe *CALLER erstellt wurden, direkt aus einem Menü oder aus einem OPM-Programm aufgerufen werden.

- *ENTMOD/*STGMDL — Der Sonderwert *ENTMOD ist der Default-Wert, der für die Aktivierungsgruppe mit dem CL-Befehl CRTPGM gesetzt werden kann.
  Der Sonderwert *STGMDL ist der Default-Wert, der mit dem CL-Befehl CRTBNDxxx angegeben werden kann, sofern das Programm nicht in der Default-Aktivierungsgruppe (Option DFTACTGRP=*NO) erstellt wird.
  Bei beiden Sonderwerten wird eine „Standard"-Aktivierungsgruppe, abhängig vom verwendeten Speichermodell (Storage Model) zugewiesen.
  Sofern das Programm im Single Level Storage Model (*SNGLVL) ausgeführt wird, wird die Aktivierungsgruppe QILE zugewiesen, im Teraspace Storage Model (*TERASPACE) die Aktivierungsgruppe QILETS.
  Diese Default-Option ist für Einsteiger gedacht, die zwar schon ILE-Konzepte verwenden, jedoch die Probleme, die bei Ausführung von ILE-Programmen in der Default-Aktivierungsgruppe auftreten können, vermeiden wollen. Die Default-Option ist auch für Einsteiger gedacht, die sich bislang noch keine Gedanken über die Verwendung und das Design von Aktivierungsgruppen gemacht haben.

3.1.10
Seite 10

3.1.10.4 Standard/Default-Aktivierungsgruppen

Bei Jobstart werden in jedem Job automatisch zwei Standard/Default-Aktivierungsgruppen gestartet.

In diesen Aktivierungsgruppen werden alle OPM-Programme aktiviert und ausgeführt. Eine der beiden Default-Aktivierungsgruppen ist für die Ausführung von Systemprogrammen reserviert, in der anderen werden die individuellen OPM-Anwendungsprogramme ausgeführt.

Diese beiden Default-Aktivierungsgruppen können nicht (manuell) beendet werden. Die einzige Möglichkeit, diese Aktivierungsgruppen zu beenden, besteht im Beenden des Jobs.

Bei der Erstellung von ILE- und Serviceprogrammen ist es nicht möglich, die Default-Aktivierungsgruppe direkt anzugeben. Die Ausführung von ILE-Programmen und insbesondere von Serviceprogrammen in der Default-Aktivierungsgruppe kann zu immensen Problemen bis hin zu Programmabbrüchen führen.

Anmerkung:

ILE- und Serviceprogramme werden bei Programmende nicht automatisch aus dem Speicher entfernt.

ILE- und Serviceprogramme können nur durch das Schließen der Aktivierungsgruppe, in der sie aktiviert sind, vollständig beendet und aus dem Speicher entfernt werden.

Die Default-Aktivierungsgruppen können auch nicht durch den Befehl RCLACTGRP beendet werden.

3.1.10.4.1 Programme und Service-Programme in Default-Aktivierungsgruppen

Aktivierungsgruppen sind Bestandteil der ILE-Konzepte und wurden mit der Einführung der ILE-Sprachen implementiert.

Um die vorhandenen OPM-Programme auch in ILE-Umgebungen unverändert ausführen zu können, wurden die Default-Aktivierungsgruppen eingeführt.

ILE- und Serviceprogramme sollten nicht in der Default-Aktivierungsgruppe ausgeführt werden, sondern in der beim Erstellen des Programms oder Serviceprogramms anzugebenden Aktivierungsgruppe. Die Default-Aktivierungsgruppe kann beim Binden von ILE- und Serviceprogrammen nicht ausgewählt werden.

Dennoch ist es möglich, ILE- und Serviceprogramme in der Default-Aktivierungsgruppe auszuführen. Werden Programme mit der Aktivierungsgruppe *CALLER erstellt und aus einem OPM-Programm aufgerufen, wird das ILE-Programm und ggf. auch die Serviceprogramme, aus denen Prozeduren aufgerufen werden, ebenfalls in der Default-Aktivierungsgruppe erstellt.

Die Ausführung von ILE-Programmen in der Default-Aktivierungsgruppe kann zu Problemen und sogar Programmabbrüchen führen.

Anmerkung:

In der Default-Aktivierungsgruppe sollten nur OPM-Programme ausgeführt werden.

Die Ausführung von ILE- und Serviceprogrammen in der Default-Aktivierungsgruppe sollte auf alle Fälle vermieden werden.

3.1.10.4.1.1 OPM-Programme in der Default-Activation-Group

Bei Programmen, die in RPGIII oder CL erstellt wurden, handelt es sich um OPM-Programme. Sie werden beim Aufruf aktiviert. Bei Programmende werden CL- und RPG-Programme mit aktivem Last-Record-Indicator (*INLR = *ON) aus dem Speicher entfernt. Mit dem Befehl RCLRSC (Ressourcen zurückfordern) können auch alle RPG-Programme, die mit RETURN beendet wurden, aus dem Speicher entfernt werden. Auch diese Programme wurden als OPM-Programme implementiert.

Um einen problemlosen Übergang von RPGIII auf RPGIV zu gewährleisten, wurden die beiden Default-Aktivierungsgruppen eingeführt, in denen alle OPM-Programme ausgeführt werden.

Die vorhandenen RPGIII-Programme konnten mit dem Befehl CVTRPGSRC (RPG-Quelle konvertieren) nach RPGIV konvertiert werden. Die konvertierte Quelle konnte ab diesem Zeitpunkt mit RPGIV weiterentwickelt werden. In der konvertierten Quelle konnten und können alle neuen Features der Programmiersprache inklusive der (Full-)Free-Format-Codierung eingesetzt werden.

Durch die Einführung der Option Standard-Aktivierungsgruppe (DFTACTGRP) in den Compile-Befehlen CRTBNDRPG und CRTBNDCL konnten die konvertierten Quellen wie die ursprünglichen RPGIII-Programme mit Option 14 (= Umwandeln) im PDM als OPM-Programme erstellt und im Anschluss in der Default-Aktivierungsgruppe ausgeführt werden.

Der einzige für den Programmierer sichtbare Unterschied waren die unterschiedlichen Kompilierungsbefehle für RPGIII und RPGIV.

Anmerkung:

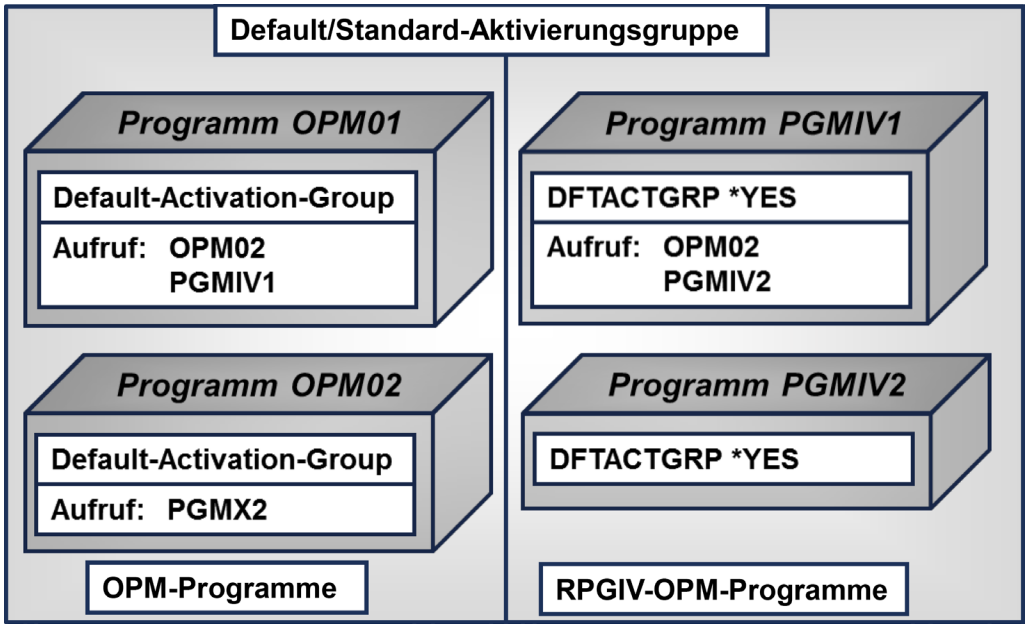
RPGIV-Programme, die mit **Standard-Aktivierungsgruppe DFTACTGRP=*YES** (Default-Wert) erstellt werden, werden weiterhin als **OPM-Programme** implementiert und in der Default-Aktivierungsgruppe ausgeführt.

Alle Programmierer, die sich bislang noch nicht mit den ILE-Konzepten auseinandergesetzt haben, jedoch für die Codierung RPGIV (inklusive Built-In-Funktionen und Free-Format) einsetzen und ihre Programme wie zuvor mit Option 14 (= Umwandeln) im PDM mit dem CL-Befehl CRTBNDRPG oder mit dem entsprechenden RDi-Äquivalent generieren, erstellen weiterhin OPM-Programme und führen sie in der Default-Aktivierungsgruppe aus.

Für RPGIV-OPM-Programme gelten die gleichen Regeln wie für RPGI-II-OPM-Programme. Werden RPGIV-OPM-Programme mit aktivem Last-Record-Indicator (*INLR = *ON) beendet, so werden sie nach Programmende komplett aus dem Speicher entfernt. RPGIV-OPM-Programme, die mit RETURN beendet werden, bleiben so lange im Speicher, bis der Befehl RCLRSC (Ressourcen zurückfordern) ausgeführt oder der Job beendet wird.

Die alten, noch nicht umgestellten OPM-Programme können problemlos die neuen oder konvertierten OPM-Programme aufrufen und wie bisher verarbeiten sowie umgekehrt.

Das folgende Schaubild zeigt mehrere alte OPM- (OPM01/OPM02) und neue RPGIV- oder CLLE- Programme (PGMIV1/PGMIV2), die mit Option DFTACTGRP=*YES erstellt wurden. Alle Programme werden in der Default-Aktivierungsgruppe aktiviert und rufen sich gegenseitig auf.

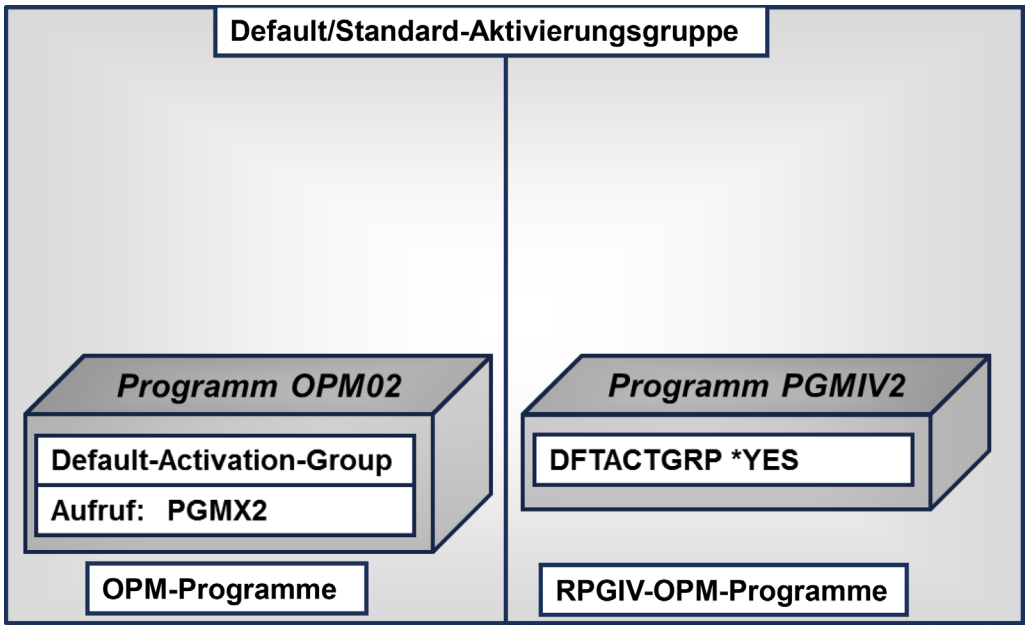


Aktivierung OPM-Programme in der Standard-Aktivierungsgruppe

Da es sich bei allen Programmen um OPM-Programme handelt, werden die Programme beim Beenden komplett aus dem Speicher entfernt, sofern der Last-Record-Indicator (*INLR=*ON) gesetzt wurde. Die OPM-Programme, die mit RETURN beendet werden, bleiben in der Default-Aktivierungsgruppe und werden erst durch Ausführen des Befehls RCLRSC bzw. bei Jobende aus dem Speicher entfernt.

Geht man davon aus, dass die Programme OPM02 und PGMIV2 jeweils mit RETURN beendet wurden, während die Programme OPM01 und PGMIV1 mit *INLR=*ON beendet werden, sieht der reservierte Speicher in der Default-Aktivierungsgruppe nach Beendigung von Programm OPM01 wie in der folgenden Abbildung aus.

OPM01 und RPGIV1 wurden beide beendet und aus dem Speicher entfernt. OPM02 und PGMIV2 wurden auf dem Stand eingefroren, den sie beim Verlassen bzw. Beenden des Programms hatten.



Default-Aktivierungsgruppe nach dem Beenden der OPM-Programme

3.1.10.4.1.2 ILE-Programme in der Default-Activation-Group

Sobald man sich näher mit den ILE-Konzepten befasst, Prozeduren und Funktionen erstellt und sie in Serviceprogrammen hinterlegt, stellt man fest, dass diese Prozeduren nicht aus OPM-Programmen aufgerufen werden können. Eine Kompilierung der rufenden Programme mit der Option DFTACTGRP=*YES schlägt fehl.

Für echte ILE-Programme ist eine zweistufige Kompilierung erforderlich, bei der zunächst die Module erstellt und anschließend in Programme gebunden werden.

Sofern nur ein einziges Modul in ein ILE-Programm gebunden werden soll, ist auch eine Kompilierung über CRTBNDxxx möglich. Die Option „Standard-Aktivierungsgruppe“ muss dazu im Compile-Befehl auf *NO gesetzt werden. Wird die Option „Standard-Aktivierungsgruppe *NO“ angegeben, wird per Default eine benannte Aktivierungsgruppe vorgeschlagen. Sofern das Programm im Teraspace ausgeführt werden soll, wird per Default die Aktivierungsgruppe QILETS verwendet, anderenfalls die Aktivierungsgruppe QILE. Wird die vorgeschlagene Aktivierungsgruppe übernommen, können ILE-Programme und Prozeduren innerhalb von Serviceprogrammen in einer einzigen Aktivierungsgruppe aktiviert und im Allgemeinen problemlos ausgeführt werden.

Sofern in dem rufenden Programm, das in einer anderen Aktivierungsgruppe (zum Beispiel der Default-Aktivierungsgruppe) ausgeführt wird, eine Dateiüberschreibung erfolgt (z. B. durch den CL-Befehl OVRDBF – Mit Datenbankdatei überschreiben oder OVRPRTF – Mit Druckerdatei überschreiben) , wird man unter Umständen feststellen, dass diese Überschreibungen in den ILE-Programmen keine Gültigkeit haben.

Weitere Probleme können in Verbindung mit der Commitment-Steuerung auftreten, das heißt Änderungen werden mit dem Befehl Commit nur teilweise festgeschrieben bzw. mit dem Befehl Rollback nur unvollständig zurückgesetzt.

Die Ursache für dieses Verhalten liegt darin, dass der Unterlassungswert für die Option OVRSCOPE (Bereich überschreiben) in den Überschreibungsbefehlen auf *ACTGRPDFN gesetzt wurde. Damit gelten Überschreibungen nur innerhalb der gleichen Aktivierungsgruppe.

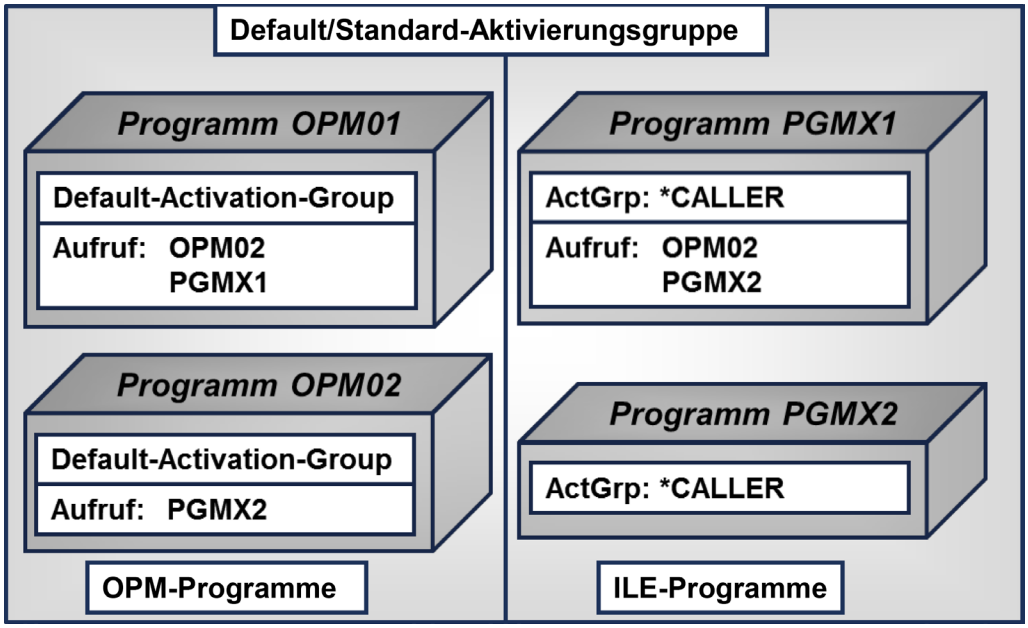
Beim CL-Befehl STRCMTCTL (Commitment-Steuerung starten) wurde der Unterlassungswert für die Option CMTSCOPE (Commit-Definitionsbereich) auf *ACTGRP gesetzt, was bedeutet, dass bei Commit oder Rollback nur die Änderungen in der aktuellen Aktivierungsgruppe festgeschrieben oder zurückgesetzt werden.

Die vermeintliche Ideallösung, ILE-Programme mit Aktivierungsgruppe *CALLER umzuwandeln, so dass alle Programme und abhängigen Serviceprogramme ebenfalls in der Default-Aktivierungsgruppe ausgeführt werden, kann jedoch in echten ILE-Umgebungen zu immensen Problemen und sogar zu Programmabbrüchen führen.

Der Grund dafür ist, dass ILE-Programme, die in der Default-Aktivierungsgruppe ausgeführt werden, anders als OPM-Programme behandelt werden. ILE-Programme und die von diesen abhängigen Serviceprogramme werden bei Verlassen des Programms nicht aus dem Speicher entfernt, auch dann nicht, wenn der Last Record Indicator (*INLR = *ON) gesetzt wurde. Beim Beenden des Programms werden lediglich die in den F-Bestimmungen definierten geöffneten Dateien geschlossen. Der reservierte Speicher wird jedoch nicht freigegeben, sondern bei erneutem Aufruf lediglich initialisiert. Werden ILE-Programme mehrfach im gleichen Job aufgerufen, sind die Folgeaufrufe auch bei aktivem Last-Record-Indicator schneller als vergleichbare OPM-Programme, da Aktivierungs-Overhead eingespart wird.

In der Default-Aktivierungsgruppe aktivierte ILE- und Serviceprogramme können nur durch das Beenden der Default-Aktivierungsgruppe freigegeben werden, also nach Jobende. Mit RCLACTGRP kann die Default-Aktivierungsgruppe nicht beendet werden, und RCLRSC entfernt keine ILE- oder Serviceprogramme. In Serviceprogrammen geöffnete Dateien werden beim RCLRSC geschlossen. Innerhalb des RPG-Codes gibt es im Nachhinein keine Möglichkeit, um festzustellen, ob eine Datei geöffnet ist oder um beim Öffnen entsprechende Fehler abzufangen.

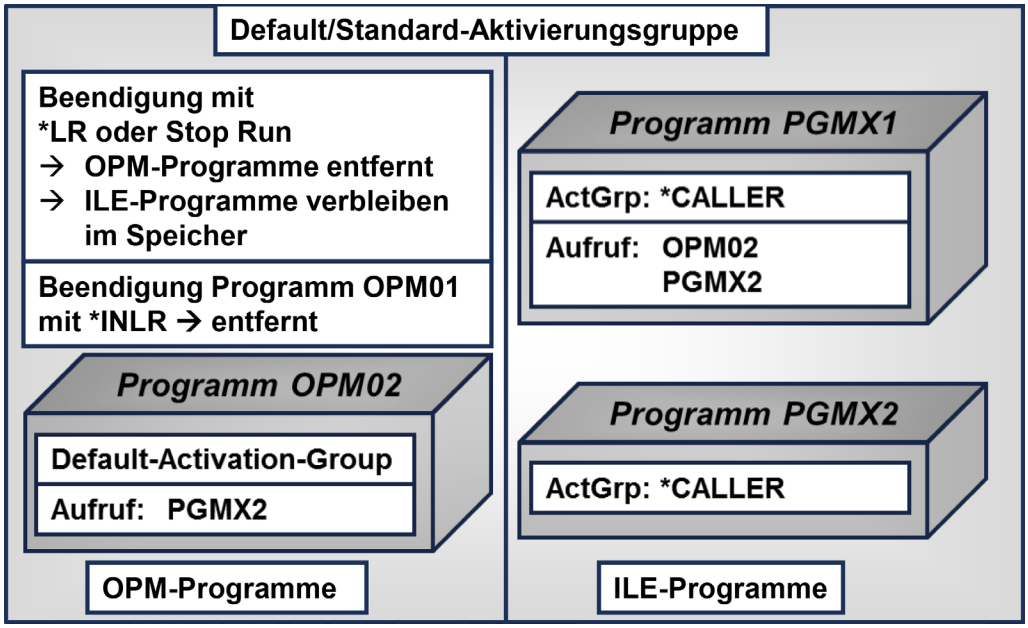
Im folgenden Beispiel werden aus dem OPM-Programm OPM01 mehrere OPM- und ILE-Programme aufgerufen. Die ILE-Programme werden ebenso wie die OPM-Programme in der Default-Aktivierungsgruppe ausgeführt, da sie mit der Option ACTGRP=*CALLER erstellt wurden und aus einem OPM-Programm aufgerufen werden.



Aktivierung von ILE-Programmen in der Default-Aktivierungsgruppe

Geht man davon aus, dass das OPM-Programm OPM01 mit aktivem Last-Record-Indicator (*INLR=*ON) und das OPM-Programm OPM02 mit RETURN beendet wird, so sieht der belegte Speicher für die Default-Aktivierungsgruppe nach Beendigung des Programms OPM01 wie folgt aus.

Das OPM-Programm OPM01 wurde komplett aus dem Speicher entfernt. Das Programm OPM02 bleibt im Speicher erhalten, da es mit RETURN beendet wird. Die ILE-Programme werden unabhängig davon, ob sie mit Last-Record-Indicator (*INLR=*ON) oder RETURN beendet werden, nicht aus dem Speicher entfernt.

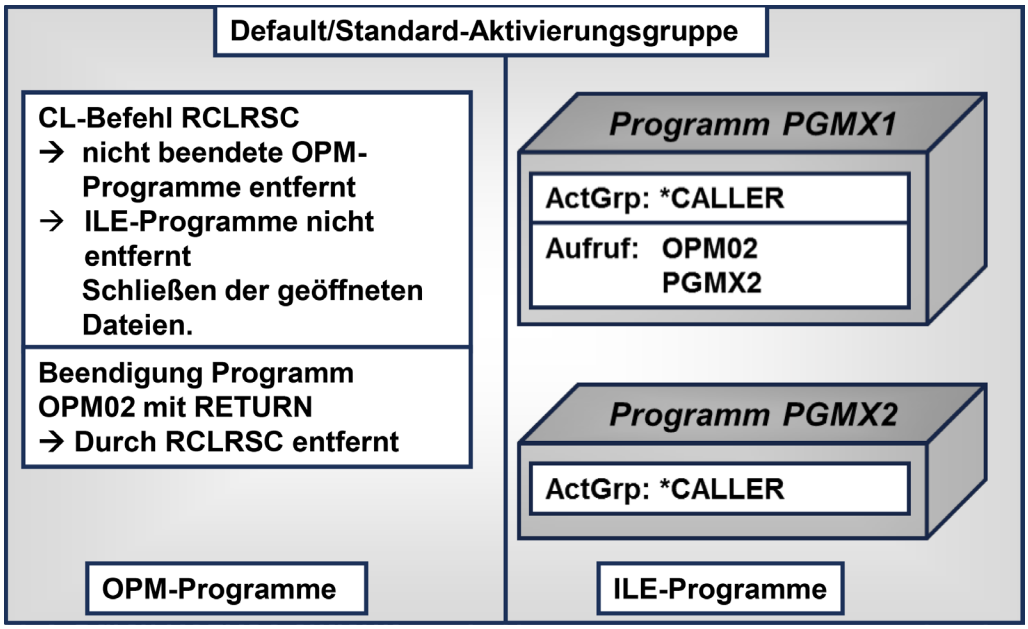


Beenden von Programmen in der Default-Aktivierungsgruppe

Wird nun versucht, den reservierten Speicher durch den CL-Befehl RCLRSC zu bereinigen, so werden alle OPM-Programme aus dem Speicher der Default-Aktivierungsgruppe entfernt, nicht jedoch die ILE-Programme und die zusammen mit den ILE-Programmen aktivierten Serviceprogramme. ILE- und Serviceprogramme bleiben so lange im Speicher, bis die Aktivierungsgruppe, in der sie aktiviert wurden, beendet wird. Da Default-Aktivierungsgruppen nicht manuell beendet werden können, bleiben die ILE- und abhängigen Serviceprogramme bis zum Jobende aktiviert.

Wird der Befehl RCLRSC ausgeführt, so werden lediglich die in der Default-Aktivierungsgruppe aktivierten ILE- und Serviceprogramme sowie die in den F-Bestimmungen definierten und nicht geschlossenen Dateien geschlossen.

Das folgende Schaubild zeigt den Speicher der Default-Aktivierungsgruppe, nachdem der CL-Befehl RCLRSC ausgeführt wurde. Das OPM-Programm OPM02 wird aus dem Speicher entfernt. Der Speicher für die beiden ILE-Programme bleibt weiterhin reserviert.



Bereinigen der Default-Activation-Group mit RCLRSC

Anmerkung:

Da der reservierte Speicher für ILE-Programme in der Default-Aktivierungsgruppe nicht bzw. nur bei Jobende freigegeben werden kann, sollten ILE-Programme nicht in der Default-Aktivierungsgruppe ausgeführt werden.

3.1.10.4.1.3 Serviceprogramme in der Default-Activation-Group

Serviceprogramme sind echte ILE-Objekte. Prozeduren aus Serviceprogrammen können aus diesem Grund nicht aus OPM-Programmen, sondern nur aus ILE-Programmen oder aus Prozeduren in anderen Serviceprogrammen aufgerufen werden.

ILE-Programme können nicht direkt in der Default-Activation-Group erstellt werden, sie können jedoch durch die Angabe der Aktivierungsgruppe *CALLER und den Aufruf aus OPM-Programmen in der Default-Aktivierungsgruppe ausgeführt werden.

Serviceprogramme können entweder mit einer benannten Aktivierungsgruppe oder mit Aktivierungsgruppe *CALLER (Unterlassungswert) erstellt werden. Wird ein Serviceprogramm mit Aktivierungsgruppe (*CALLER) erstellt, so wird es in der gleichen Aktivierungsgruppe wie dasjenige Programm aktiviert, das irgendeine Prozedur aus dem Serviceprogramm aufruft. Wird jetzt eine Prozedur aus einem Serviceprogramm aufgerufen, das mit Aktivierungsgruppe *CALLER erstellt und in der Default-Aktivierungsgruppe aktiviert wurde, so wird auch das Serviceprogramm in der Default-Aktivierungsgruppe ausgeführt. Die Aktivierung von Serviceprogrammen in der Default-Aktivierungsgruppe kann zu immensen Problemen führen, insbesondere, wenn der CL-Befehl RCLRSC ausgeführt wurde, um Speicher freizugeben.

Bei der Ausführung des Befehls RCLRSC (Ressourcen zurückfordern) werden alle in den F-Bestimmungen der ILE- und Serviceprogramme geöffneten Dateien geschlossen.

Beim erneuten Aufruf einer Prozedur in einem Serviceprogramm, in der auf Dateien, die in F-Bestimmungen definiert wurden, zugegriffen wird, erfolgt ein Abbruch mit der Fehlermeldung MCH3601 (Zeiger für angegebene Position nicht gesetzt). Aktuell gibt es keine Möglichkeit, um diesen Fehler abzufangen, weder durch die Built-In-Function %OPEN noch durch die Erweiterung (E) bei Operation-Codes wie OPEN, READ, WRITE in Verbindung mit der Built-In-Function %ERROR.

Da das Serviceprogramm in der Default-Aktivierungsgruppe aktiviert wurde, kann es auch nicht geschlossen oder aus dem Speicher entfernt werden. Bleibt nur, den Job zu beenden und neu zu starten.

Anmerkung:

Die Aktivierung von Serviceprogrammen in der Default-Aktivierungsgruppe kann zu massiven Problemen bis hin zu Abbrüchen führen.

Deshalb sollte eine Aktivierung von Serviceprogrammen in der Default-Aktivierungsgruppe unter allen Umständen vermieden werden.

Da die Aktivierung von Serviceprogrammen in der Default-Aktivierungsgruppe nur durch einen Aufruf aus ILE-Programmen heraus erfolgen kann, die ihrerseits in der Default-Aktivierungsgruppe aktiviert sind, sollten auch ILE-Programme nicht in der Default-Aktivierungsgruppe ausgeführt werden.

Die Entscheidung, ILE-Programme mit Aktivierungsgruppe *CALLER zu erstellen, sollte deshalb bestens durchdacht sein.

3.1.10.5 Aktivierungsgruppe *NEW

Sofern ILE-Programme in der Default-Aktivierungsgruppe oder einer benannten Aktivierungsgruppe aktiviert wurden, bleiben sie nach Beendigung weiterhin im Speicher, um den Aktivierungs-Overhead bei einem Folgeaufruf zu minimieren. ILE-Programme, die in der Default-Aktivierungsgruppe aktiviert wurden, können nur durch Beendigung des Jobs freigegeben werden.

Es gibt jedoch Programme, die nur einmalig ausgeführt werden. Nach Beendigung dieser Programme sollten alle zur Ausführung des Programms notwendigen Elemente (zum Beispiel Dateien, Serviceprogramme, offene Datenpfade) spurlos aus dem Speicher verschwinden. Um dieser Anforderung gerecht zu werden, kann beim Erstellen des Programms für die Aktivierungsgruppe der Sonderwert *NEW angegeben werden.

Anmerkung:

Serviceprogramme können im Gegensatz zu Programmen nicht direkt mit Aktivierungsgruppe *NEW erstellt werden.

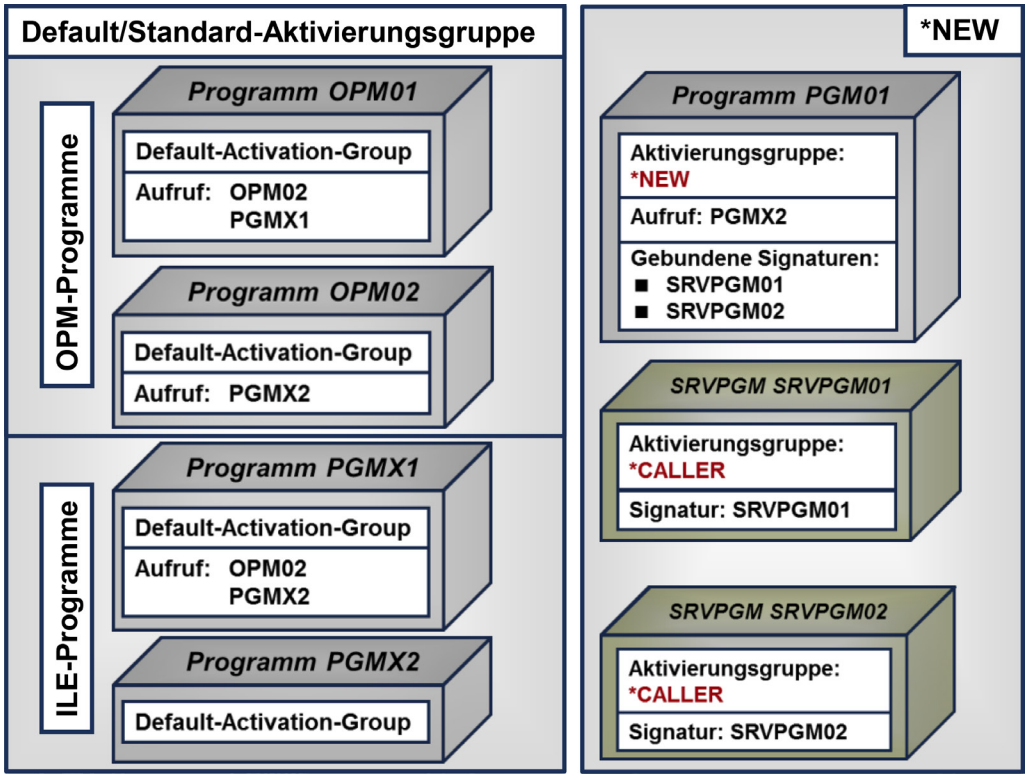
Durch die Erstellung der Serviceprogramme mit Aktivierungsgruppe *CALLER, können sie auch in der neuen Aktivierungsgruppe des rufenden Programms ausgeführt werden.

Bei jedem Aufruf eines Programms mit Aktivierungsgruppe *NEW wird eine neue Aktivierungsgruppe generiert und das Programm sowie alle abhängigen zum Beispiel mit Aktivierungsgruppe *CALLER erstellten Programme bzw. Serviceprogramme werden in dieser Aktivierungsgruppe ausgeführt. Bei Programmende werden das Programm sowie alle abhängigen Programme, Serviceprogramme und offenen Datenpfade (ODP) aus dem Speicher entfernt.

RPG-Programme, die dem Zyklus unterliegen bzw. eine klassische/zyklische Main-Procedure haben, können normalerweise nicht rekursiv aufgerufen werden. Dies war zu Zeiten der Green-Screen-Programmierung oft ein Problem. So konnten zum Beispiel über Funktionstasten unterschiedliche Programme mit unterschiedlichsten Funktionen aktiviert werden. Da diese Programme sich auch gegenseitig aufrufen konnten, konnte es vorkommen, dass zum Zeitpunkt x das Ausgangsprogramm erneut aufgerufen wurde. Um hierbei einen durch diesen RPG-Zyklus verursachten Abbruch zu vermeiden, mussten einige Klimmzüge gemacht werden, wie etwa Call Stack abgreifen und auf rufendes Programm zurückspringen oder ein Duplikat des Ausgangsprogramms erstellen.

Erstellt man ein Programm mit Aktivierungsgruppe *NEW, so kann dieses Programm auch mit einer zyklischen Main-Procedure rekursiv aufgerufen werden. Der Grund hierfür liegt darin, dass das Programm bei jedem Aufruf in einer neuen Aktivierungsgruppe aktiviert wird. Dadurch wird intern ein Duplikat des Programms bzw. der entsprechenden Ressourcen erstellt. Das so erzeugte Duplikat kann völlig unabhängig vom rufenden Programm ausgeführt werden.

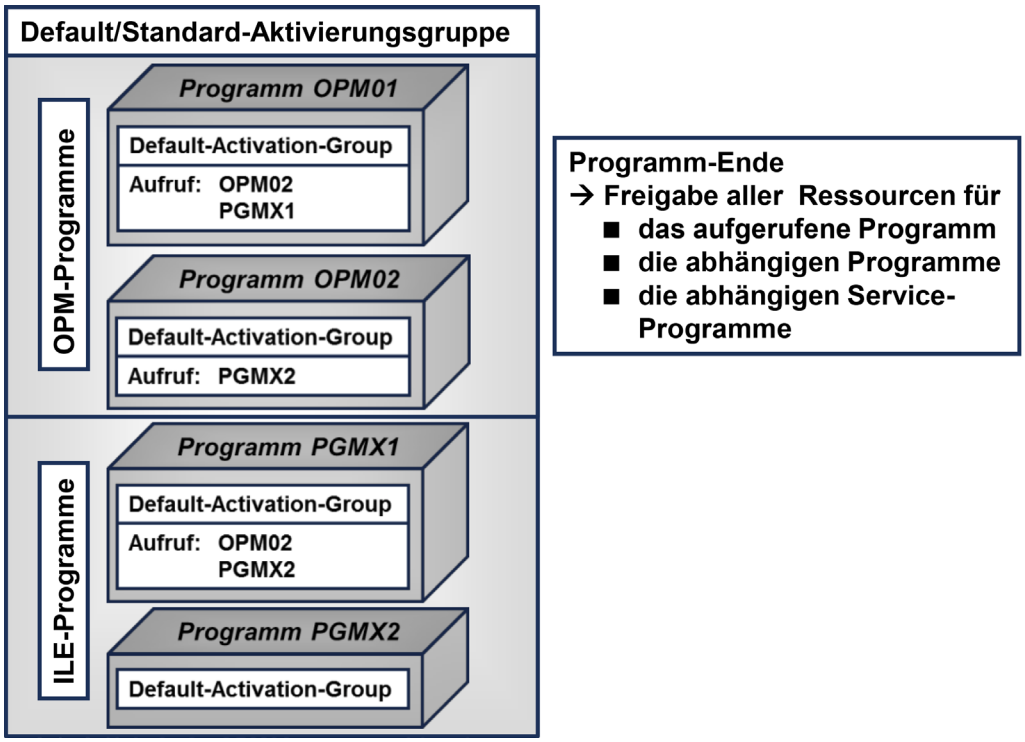
Das folgende Beispiel zeigt den Speicherbereich eines Jobs. In diesem Speicher befinden sich die Default-Aktivierungsgruppen, in denen die aktiven System- sowie OPM-Programme und einige ILE-Programme aktiviert sind. Des Weiteren wurde eine neue Aktivierungsgruppe gebildet, in der das Programm PGM01 aktiviert wurde. Da aus diesem Programm diverse Prozeduren aufgerufen werden, die in den Serviceprogrammen SRVPGM01 und SRVPGM02 hinterlegt sind, und diese Serviceprogramme mit Aktivierungsgruppe *CALLER erstellt wurden, wurden diese Serviceprogramme in der neuen Aktivierungsgruppe ebenfalls aktiviert.



*Aktivierte Aktivierungsgruppe *NEW*

3.1.10
Seite 24

Sobald das Programm PGM01 beendet wird, wird alles, was zusammen mit dem Programm in der neuen Aktivierungsgruppe aktiviert wurde, wieder aus dem Speicher entfernt. Im aktuellen Beispiel verbleiben nach Beendigung von PGM01 nur die Default-Aktivierungsgruppen mit den aktiven OPM-Programmen sowie die aktiven oder bereits beendeten ILE-Programme im Jobspeicher.



*Jobspeicher nach Beendigung eines Programms mit Aktivierungsgruppe *NEW*

3.1.10.6 Aktivierungsgruppe *CALLER

Serviceprogramme können im Gegensatz zu Programmen weder in der Default-Aktivierungsgruppe noch in einer neuen Aktivierungsgruppe aktiviert werden. Beim Binden von Serviceprogrammen können lediglich die Aktivierungsgruppe *CALLER oder eine benannte Aktivierungsgruppe angegeben werden.

Serviceprogramme werden in der Regel erstellt, um wiederverwendbare Prozeduren und Funktionen bereitzustellen, ohne dass die entsprechenden Module in mehrere Programme gebunden werden müssen. Da Serviceprogramme Prozeduren und Funktionen enthalten, die für den Programmablauf erforderlich sind und deshalb zusammen mit dem rufenden Programm aktiviert werden sollten, werden sie per Default in Aktivierungsgruppe *CALLER erstellt.

Damit wird sichergestellt, dass das mit *CALLER erstellte Serviceprogramm zusammen mit dem rufenden Programm – unabhängig davon, in welcher Aktivierungsgruppe dieses läuft – aktiviert und als Einheit ausgeführt wird und auch zusammen mit dem rufenden Programm beendet werden kann.

Der Sonderwert *CALLER für Aktivierungsgruppen kann auch beim Erstellen von Programmen angegeben werden, die in der gleichen Aktivierungsgruppe wie das rufende Programm ausgeführt werden sollen.

Achtung:

Bei der Erstellung von Programmen mit Aktivierungsgruppe *CALLER ist Vorsicht geboten.

Werden ILE-Programme, die mit *CALLER erstellt werden, aus OPM-Programmen aufgerufen, werden sie in der Default-Aktivierungsgruppe ausgeführt.

Werden aus diesen Programmen wiederum Prozeduren aus Service-Programmen aufgerufen, die ebenfalls mit Aktivierungsgruppe *CALLER erstellt wurden, so werden diese Serviceprogramme ebenfalls in der Default-Aktivierungsgruppe ausgeführt, was zu Problemen bis hin zu Abbrüchen führen kann.

Werden die rufenden Programme in unterschiedlichen Aktivierungsgruppen ausgeführt, zum Beispiel in einer neuen oder auch in einer benannten Aktivierungsgruppe, so wird das (Service-)Programm mehrfach innerhalb des gleichen Jobs aktiviert.

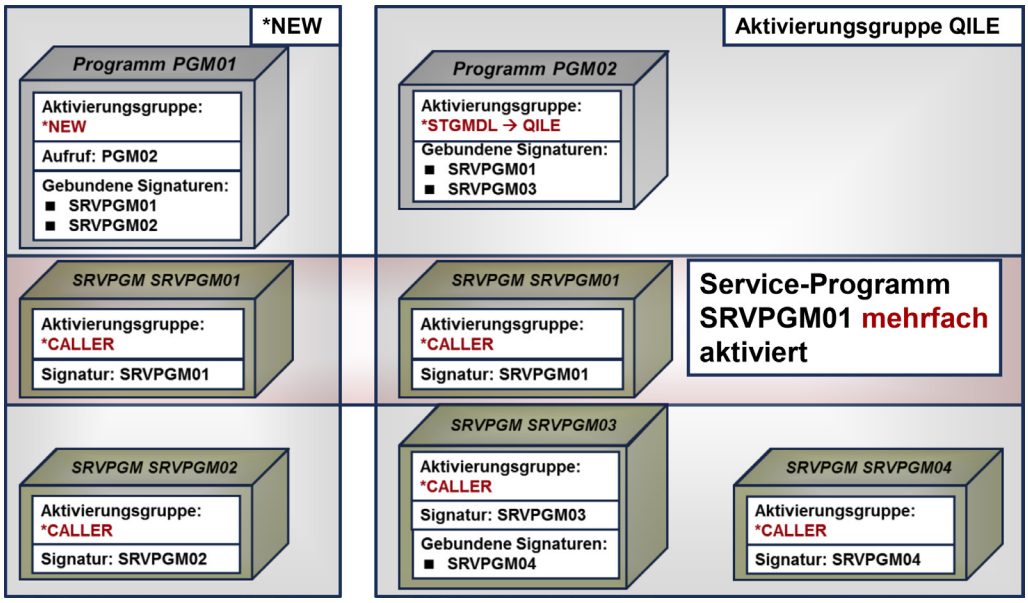
Eine Mehrfachaktivierung innerhalb des gleichen Jobs kann durchaus sinnvoll sein, etwa für alle Prozeduren, in denen Datensätze erstellt, geändert oder gelöscht werden. Diese Prozeduren werden zusammen mit dem rufenden Programm aktiviert und als Einheit ausgeführt.

Im folgenden Beispiel wird das Programm PGM01 in einer neuen Aktivierungsgruppe ausgeführt. Aus dem Programm werden Prozeduren in den Serviceprogrammen SRVPGM01 und SRVPGM02 ausgeführt. Da beide Serviceprogramme mit Aktivierungsgruppe *CALLER erstellt wurden, wurden sie ebenfalls in der neuen Aktivierungsgruppe aktiviert.

Aus Programm PGM01 wird außerdem das Programm PGM02 aufgerufen, das in der Aktivierungsgruppe QILE aktiviert wird. Aus diesem Programm werden wiederum Prozeduren aufgerufen, die in den Serviceprogrammen SRVPGM01 und SRVPGM03 hinterlegt sind. Beide Serviceprogramme wurden mit Aktivierungsgruppe *CALLER erstellt und somit ebenfalls in der Aktivierungsgruppe QILE aktiviert.

Aus Serviceprogramm SRVPGM03 werden außerdem Prozeduren aus Serviceprogramm SRVPGM04 aufgerufen. Da das Serviceprogramm SRVPGM04 mit Aktivierungsgruppe *CALLER erstellt wurde, wird es ebenfalls in der Aktivierungsgruppe QILE aktiviert.

Sowohl aus PGM01 als auch aus PGM02 wurden Prozeduren aus Serviceprogramm SRVPGM01 aufgerufen. Da das Serviceprogramm SRVPGM01 mit Aktivierungsgruppe *CALLER erstellt wurde und die beiden rufenden Programme in unterschiedlichen Aktivierungsgruppen aktiviert wurden, wurde auch das Serviceprogramm SRVPGM01 doppelt aktiviert – in der neuen Aktivierungsgruppe und in Aktivierungsgruppe QILE.

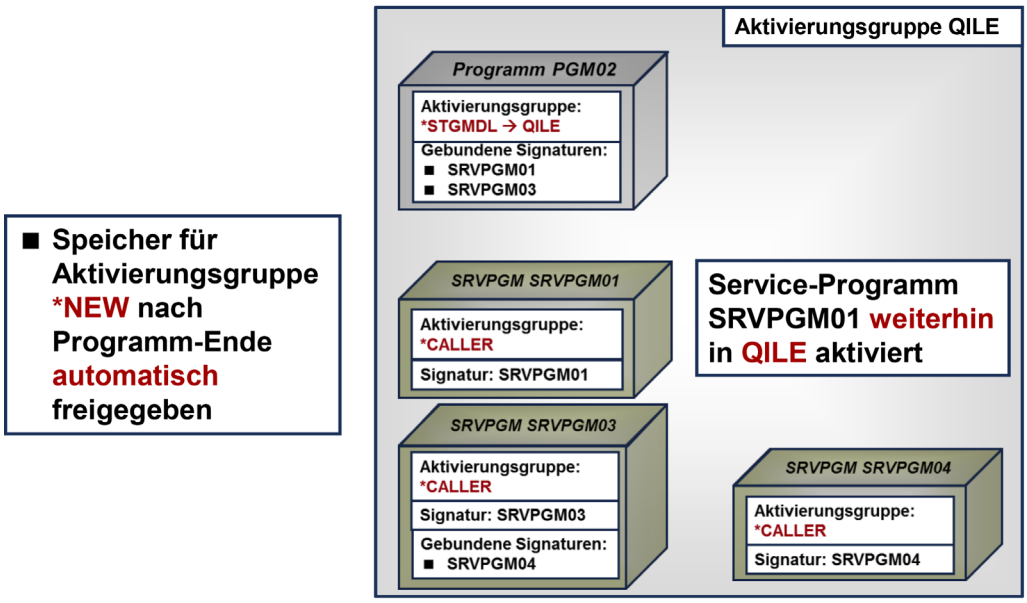


*Aktivierung von (Service-)Programmen mit Aktivierungsgruppe *CALLER*

Wird PGM01 beendet, wird nicht nur das Programm selber aus dem Speicher entfernt, sondern auch die in der gleichen Aktivierungsgruppe aktivierten Serviceprogramme SRVPGM01 und SRVPGM02. Die in der Aktivierungsgruppe QILE aktivierten Programme und Serviceprogramme bleiben weiterhin aktiviert.

3.1.10
Seite 28

Die folgende Abbildung zeigt den Speicher nach Beendigung von Programm PGM01:



Beendigung von Programmen und abhängigen Serviceprogrammen

Wird Programm PGM01 erneut aufgerufen, muss auch das Serviceprogramm SRVPGM01 in der neuen Aktivierungsgruppe erneut aktiviert werden. Das in der Aktivierungsgruppe QILE aktivierte Serviceprogramm SRVPGM01 kann nicht genutzt werden.

3.1.10.7 Benannte Aktivierungsgruppen

Bei echter modularer Programmierung wird der Quell-Code für ein Programm auf ein Minimum reduziert. Im Idealfall werden nur Prozeduren, die sich in diversen Serviceprogrammen befinden, in der vorgegebenen Reihenfolge ausgeführt.

Einige der aufgerufenen Prozeduren müssen zusammen mit dem Programm als Einheit ausgeführt werden, während andere Prozeduren allgemeingültig sind und auch von anderen Programmen und Prozeduren genutzt werden können.

Werden die Programme mit der Aktivierungsgruppe *NEW erstellt, wird bei jedem Aufruf eine neue Aktivierungsgruppe generiert, in der allen gebundenen und mit Aktivierungsgruppe *CALLER erstellten Serviceprogramme aktiviert werden. Bei Programmende wird der für die Ausführung des Programms reservierte Speicher inklusive dem für die Serviceprogramme reservierten Bereich wieder komplett freigegeben. Bei einem erneuten Programmaufruf muss wiederum eine komplette Aktivierung erfolgen, was einiges an Overhead verursacht.

Werden Programme mit der Aktivierungsgruppe *CALLER erstellt, so werden sie in der gleichen Aktivierungsgruppe wie das rufende Programm bzw. die rufende Prozedur aktiviert. Bei Programmende bleibt das Programm im Speicher. Der Speicher kann nur durch das Schließen der Aktivierungsgruppe freigegeben werden, in der das rufende Programm bzw. die Prozedur ausgeführt wird. Wird das Programm von einem OPM-Programm aufgerufen, wird es in der Default-Aktivierungsgruppe aktiviert. Da die Default-Aktivierungsgruppe nur bei Jobende den reservierten Speicher freigibt, kann ein ILE- oder Serviceprogramm, das in der Default-Aktivierungsgruppe ausgeführt wird, nicht deaktiviert werden. Insbesondere die Ausführung von Serviceprogrammen in der Default-Aktivierungsgruppe kann zu Problemen führen.

3.1.10
Seite 30

Die beste Lösung, um ILE- oder Serviceprogramme gezielt zu aktivieren oder zu deaktivieren, sind benannte Aktivierungsgruppen. Dabei wird der Name der Aktivierungsgruppe, in der das (Service-)Programm ausgeführt werden soll, beim Erstellen bzw. Binden des (Service-)Programms fix angegeben.

Der Name der Aktivierungsgruppe ist frei wählbar und wird nicht daraufhin geprüft, ob ein anderes (Service-)Programm in der gleichen Aktivierungsgruppe aktiviert werden soll.

Beim Aufruf des Programms werden die direkt gebundenen Serviceprogramme per Default ebenfalls aktiviert. Gebundene Serviceprogramme, die mit Aktivierungsgruppe *CALLER aktiviert wurden, werden in der gleichen Aktivierungsgruppe aktiviert wie das Programm. Serviceprogramme mit einer benannten Aktivierungsgruppe werden in dieser benannten Aktivierungsgruppe aktiviert.

Programme, die in der gleichen benannten Aktivierungsgruppe ausgeführt werden, werden erst mit dem ersten Aufruf des Programms aktiviert.

Seit Release 6.1 ist es möglich auch die Aktivierung von Serviceprogrammen bis zum ersten Aufruf einer Prozedur aus diesem Serviceprogramm zu verzögen. In hochmodularen Anwendungen, bei denen viele Prozeduren in vielen unterschiedlichen Serviceprogrammen hinterlegt sind, die in diversen Aktivierungsgruppen ausgeführt werden, kann es vorkommen, dass bestimmte Serviceprogramme bis zum Ende des Jobs nicht angesprochen werden. Durch die Möglichkeit, die Aktivierung zu verzögern, kann unnötiger Aktivierungsaufwand vermieden werden.

Wird die Aktivierungsgruppe über RCLACTGRP (Aktivierungsgruppe zurückfordern) geschlossen, werden alle in der Aktivierungsgruppe aktivierten (Service-)Programme aus dem Speicher entfernt.

3.1.10.7.1 Programme in benannten Aktivierungsgruppen

ILE-Programme bzw. Programme, aus denen in Serviceprogrammen hinterlegte Prozeduren aufgerufen werden, können nicht direkt in der Default-Aktivierungsgruppe ausgeführt werden.

Werden ILE-Programme mit dem Befehl CRTBNDxxx (Binderprogramm erstellen) generiert, muss die Option „Default-Aktivierungsgruppe“ (DFTACTGRP) auf *NO gesetzt werden.

Bei der Auswahl Default-Aktivierungsgruppe *NO ist die Angabe einer Aktivierungsgruppe erforderlich. Bei Verwendung des Default-Werts für die Aktivierungsgruppe wird das Programm abhängig vom verwendeten Speichermodell (Single Level oder Teraspace) entweder in der Aktivierungsgruppe QILE (Single Level) oder QILETS (Teraspace) ausgeführt.

Bei der zweitstufigen Kompilierung bzw. dem Binden über den Befehl CRTPGM ist der Unterlassungswert für die Aktivierungsgruppe *ENDMOD. Bei Verwendung dieses Unterlassungswerts werden reine RPGLE-, CBLLE- und CLLE-Programme – abhängig vom verwendeten Speichermodell – entweder in der Aktivierungsgruppe QILE (Single Level) oder QILETS (Teraspace) ausgeführt. Für alle anderen ILE-Programme (zum Beispiel CLE, ILE C) wird per definitionem Aktivierungsgruppe *NEW verwendet.

Anmerkung:

Durch die Verwendung von benannten Aktivierungsgruppen wird verhindert, dass ILE-Programme und die zur Ausführung erforderlichen Serviceprogramme beim Aufruf durch ein OPM-Programm in der Default-Aktivierungsgruppe aktiviert werden.

Probleme, die durch die Aktivierung von ILE- und Serviceprogrammen in der Default-Aktivierungsgruppe auftreten können, werden dadurch vermieden.

Sofern die Programme in einer der Aktivierungsgruppen QILE oder QILETS oder in immer der gleichen benannten Aktivierungsgruppe ausgeführt werden, werden die gebundenen Serviceprogramme, die mit Aktivierungsgruppe *CALLER erstellt wurden, ebenfalls in dieser Aktivierungsgruppe ausgeführt.

Bei Programmaufruf wird geprüft, ob die gebundenen Serviceprogramme bereits in der Aktivierungsgruppe aktiviert wurden. Ist das Serviceprogramm bereits in der Aktivierungsgruppe vorhanden, erfolgt keine erneute Aktivierung. Stattdessen wird das aktivierte Serviceprogramm verwendet. Wird nur eine einzige Aktivierungsgruppe verwendet, werden alle (Service-)Programme nur einmalig aktiviert.

Bei diesem Verfahren ist es jedoch nicht möglich, den Speicher gezielt für einzelne (Service-)Programme freizugeben. Die Freigabe des Speichers kann nur durch das Schließen der Aktivierungsgruppe erfolgen, wodurch alle in der Aktivierungsgruppe aktiven (Service-)Programme deaktiviert werden.

Bei Verwendung der Aktivierungsgruppen QILE oder QILETS sollte man bedenken, dass auch Fremdsoftware in diesen Aktivierungsgruppen ausgeführt werden könnte. Werden diese Aktivierungsgruppen mit RCLACTGRP (Aktivierungsgruppe zurückfordern) beendet, werden unter Umständen (Service-)Programme deaktiviert, die nicht hätten freigegeben werden dürfen.

Anmerkung:

Um Konflikte bei der Freigabe eigener ILE-Objekte und von ILE-Objekten aus Fremdsoftware zu vermeiden, sollten die eigenen ILE-Programme nicht in den Aktivierungsgruppen QILE oder QILETS ausgeführt werden. Stattdessen sollte man eine eigene benannte Aktivierungsgruppe (beispielsweise Name der Firma) verwenden.

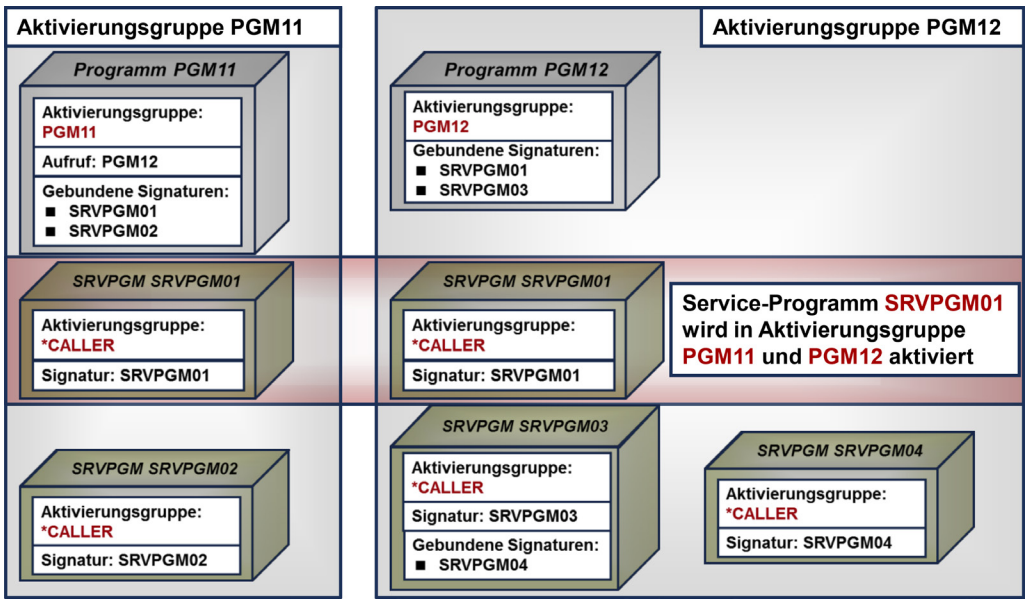
Sofern Programme nur einmalig ausgeführt werden oder wenn Programme mit gebundenen Serviceprogrammen gezielt aus dem Speicher entfernt werden sollen, müssen unterschiedliche Aktivierungsgruppen verwendet werden. Am einfachsten ist es, jedes Programm, das aus einem Menü aufgerufen wird, das submittet oder über einen Schedular-Eintrag aktiviert wird, in einer eigenen Aktivierungsgruppe auszuführen. Bei der Erstellung könnte man zum Beispiel den Programmnamen als Aktivierungsgruppennamen angeben.

Die Programme können nach Beendigung im Speicher verbleiben oder jederzeit gezielt entfernt werden, indem die Aktivierungsgruppe mit dem Programmnamen über den Befehl RCLACTGRP (Aktivierungsgruppe zurückfordern) beendet wird.

In der folgenden Abbildung wird Programm PGM11 in der Aktivierungsgruppe PGM11 (Programm = Aktivierungsgruppenname) gestartet. Die beiden Serviceprogramme SRVPGM01 und SRVPGM02, die mit Aktivierungsgruppe *CALLER erstellt wurden, werden ebenfalls in Aktivierungsgruppe PGM11 aktiviert.

Sobald das Programm PGM12 aus PGM11 aufgerufen wird, wird die Aktivierungsgruppe PGM12 erstellt, in der das Programm PGM12 sowie die Serviceprogramme SRVPGM01, SRVPGM03 und SRVPGM04 aktiviert werden. SRVPGM04 wurde ebenfalls mit der Aktivierungsgruppe *CALLER generiert. Da in SRVPGM04 mindestens eine Prozedur aus SRVPGM03 aufgerufen wird, muss Serviceprogramm SRVPGM04 zusammen mit Serviceprogramm SRVPGM03 aktiviert werden.

Serviceprogramm SRVPGM01 wird in diesem Beispiel mehrfach, das heißt in beiden Aktivierungsgruppen aktiviert.



Benannte Aktivierungsgruppen (Programm = Aktivierungsgruppenname)

Werden PGM11 oder PGM12 beendet, bleiben sie mit allen abhängigen Serviceprogrammen im Speicher so lange existent, bis die Aktivierungsgruppen durch das Ausführen des Befehls RCLACTGRP (Aktivierungsgruppe zurückfordern) beendet wird. Da bei Folgeaufrufen keine erneute Aktivierung erfolgen muss und Aktivierungs- und Initialisierungsaufwand entfällt, können sie annähernd so schnell aufgerufen werden wie gebundene Prozeduren. Im Bedarfsfall können die Programme gezielt durch das Schließen der Aktivierungsgruppen aus dem Speicher entfernt werden.

Unterprogramme, die aus diesen Programmen aufgerufen werden, und zusammen mit dem rufenden Programm als Einheit ausgeführt werden sollen, können wahlweise mit der hartcodierten Aktivierungsgruppe (ACTGRP = rufender Programmname) oder mit Aktivierungsgruppe *CALLER erstellt werden.

Sofern allerdings die Gefahr besteht, dass diese Unterprogramme aus OPM-Programmen aufgerufen werden könnten, sollte der Aktivierungsgruppenname direkt angegeben werden, um eine Ausführung in der Default-Aktivierungsgruppe zu vermeiden.

Sowohl in CL als auch in RPG können die Aktivierungsgruppe und andere ILE-Spezifikationen direkt im Kompilierungsbefehl CRTBNDXXX angegeben werden. Die Gefahr, dass man vergisst den Default-Wert zu überschreiben bzw. eine falsche Aktivierungsgruppe angibt, kann vermieden werden, indem die entsprechenden Informationen direkt im Quell-Code hinterlegt werden.

Im CLLE-Quell-Code können die Default-Aktivierungsgruppe und die tatsächliche Aktivierungsgruppe über den Befehl DCLPRCOPT (Verarbeitungsdeklarationen deklarieren) gesetzt werden.

Im folgenden Beispiel werden im CL-Code die Aktivierungsgruppe und die im Binderschritt erforderlichen Binderverzeichnisse hinterlegt.

```
     DCLPRCOPT     DFTACTGRP(*NO) +
                      ACTGRP(MYPGMGRP) +
                      BNDDIR(MYBNDDIR1 MYBNDDIR2)
```

Aktivierungsgruppe und Binderverzeichnisse im CL-Quell-Code hinterlegen

In RPG stehen die Schlüsselworte DFTACTGRP und ACTGRP, die in den H-Bestimmungen angegeben werden können, zur Verfügung (vgl. Kapitel 2.6.6.).

Anmerkung:

Idealerweise sollten aus Programmen keine weiteren Programme, sondern lediglich Prozeduren in Serviceprogrammen aufgerufen werden.

In gewachsenen Anwendungen ist dies jedoch nicht immer gleich von Anfang an möglich, sollte für die Zukunft jedoch anvisiert werden.

3.1.10.7.2 Serviceprogramme in benannten Aktivierungsgruppen

Fährt man die Strategie, dass jedes Programm seine eigene Aktivierungsgruppe erhält, so werden in diesen Aktivierungsgruppen alle Serviceprogramme aktiviert, aus denen Prozeduren aufgerufen werden und die mit Aktivierungsgruppe *CALLER erstellt wurden.

Das hat zur Folge, dass das gleiche Serviceprogramm für mehrere Programme – die in unterschiedlichen Aktivierungsgruppen ausgeführt werden – aktiviert wird.

Bei vielen Programmen und wenigen großen Serviceprogrammen kann es durch die mehrfach aktivierten Serviceprogramme zu Speicherproblemen kommen.

Werden die wenigen großen Serviceprogramme in mehrere kleine Serviceprogramme konvertiert, ist der Aktivierungsaufwand unter Umständen höher, da anstatt eines einzigen Serviceprogramms mehrere Serviceprogramme lokalisiert und in den Speicher geladen werden müssen. Der Speicherverbrauch in Summe ist jedoch niedriger, da nicht immer alle vorhandenen Serviceprogramme aktiviert werden müssen.

Es stellt sich jedoch die Frage, ob es wirklich erforderlich ist, die Serviceprogramme zusammen mit den rufenden Programmen in der gleichen Aktivierungsgruppe zu aktivieren.

Für Serviceprogramme, die nur universell einsetzbare Prozeduren enthalten – etwa Datumsfunktionen, String-Funktionen, Währungsumrechnungsprozeduren etc. – muss diese Frage eindeutig mit NEIN beantwortet werden. Anstatt zum Beispiel das Serviceprogramm, das die Datumsrechnungsfunktionen enthält, für jedes Programm separat zu aktivieren, kann dieses Serviceprogramm in seiner eigenen benannten Aktivierungsgruppe einmalig aktiviert werden und von allen Programmen und anderen Serviceprogrammen im gleichen Job genutzt werden.

Auf der anderen Seite gibt es Serviceprogramme, die Prozeduren enthalten, die unbedingt mit dem rufenden Programm zusammen aktiviert werden sollten. Dazu gehören Serviceprogramme, die spezielle, programmspezifische Prozeduren enthalten und die nur erstellt wurden, um das Binden von Modulen zu Programmen zu umgehen und die Prozeduren im Bedarfsfall auch anderen Programmen zugänglich zu machen.

Des Weiteren sollten Serviceprogramme, in denen Informationen auf Programmebene verwaltet werden, zusammen mit dem Programm aktiviert werden. In solchen Serviceprogrammen werden Informationen in globalen Datenstrukturen gespeichert, die nur durch den Aufruf von Prozeduren geändert werden können. Bei solchen Serviceprogrammen wird in Grenzen objektorientiertes Verhalten imitiert.

Serviceprogramme, die Insert, Update oder Delete-Prozeduren auf Datensätze enthalten, sollten ebenfalls mit Aktivierungsgruppe *CALLER erstellt und zusammen mit dem rufenden Programm aktiviert werden. Aktivierungsgruppe *CALLER ist für Serviceprogramme mit Insert/Update oder Delete-Funktionen wichtig, insbesondere dann, wenn mit Commitment Control gearbeitet wird. Der Unterlassungswert für den Commit-Definitionsbereich (CMTSCOPE) im Befehl STRCMTCTL (Commit-Steuerung starten) ist mit *ACTGRP hinterlegt, das heißt, Commit schreibt nur die Änderungen innerhalb der Aktivierungsgruppe fest, während Rollback nur die Änderungen innerhalb der Aktivierungsgruppe zurücksetzt.

Im folgenden Beispiel wird das Programm PGM01 in Aktivierungsgruppe PGM01 aktiviert. Über dieses Programm werden Funktionen aus den Serviceprogrammen SRVPGM01 und SRVPGM02 aufgerufen.

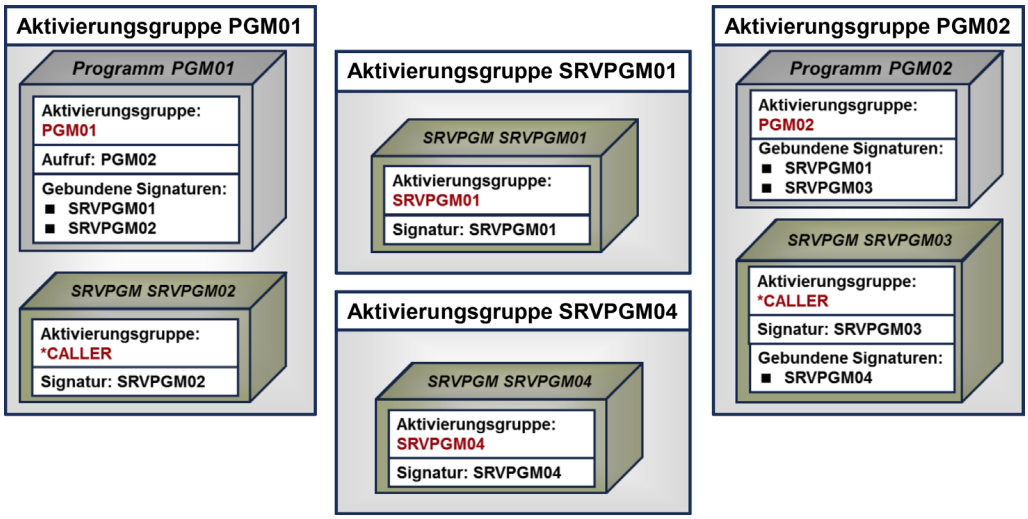
Das Serviceprogramm SRVPGM02 wurde mit Aktivierungsgruppe *CALLER erstellt und wird deshalb in der Aktivierungsgruppe PGM01 aktiviert.

Das Serviceprogramm SRVPGM01 enthält allgemeingültige Prozeduren und wurde deshalb mit einer eigenen Aktivierungsgruppe (Serviceprogrammname = Aktivierungsgruppenname = SRVPGM01) erstellt und auch in dieser Aktivierungsgruppe aktiviert.

Aus Programm PGM01 wird außerdem das Programm PGM02 aufgerufen. Sobald der Aufruf dieses Programms erfolgt, wird es in der benannten Aktivierungsgruppe PGM02 aktiviert. Aus dem Programm PGM02 werden wiederum Funktionen aufgerufen, die in den Serviceprogrammen SRVPGM01 und SRVPGM02 hinterlegt sind.

Serviceprogramm SRVPGM01 wird in Aktivierungsgruppe SRVPGM01 ausgeführt. Da diese Aktivierungsgruppe bereits durch das Programm PGM01 erstellt und Serviceprogramm SRVPGM01 in dieser Aktivierungsgruppe aktiviert wurde, ist eine erneute Aktivierung des Serviceprogramms SRVPGM01 nicht erforderlich. Stattdessen kann das bereits aktivierte Serviceprogramm verwendet werden.

SRVPGM02 wurde mit Aktivierungsgruppe *CALLER erstellt und deshalb in Aktivierungsgruppe PGM02 aktiviert. Aus Serviceprogramm SRVPGM02 werden Prozeduren aus Serviceprogramm SRVPGM04 aufgerufen, das wiederum mit einer benannten Aktivierungsgruppe SRVPGM04 erstellt wurde. Dadurch wird beim Aufruf von PGM02 auch die benannte Aktivierungsgruppe SRVPGM04 erstellt, in der das Serviceprogramm SRVPGM04 aktiviert wird.

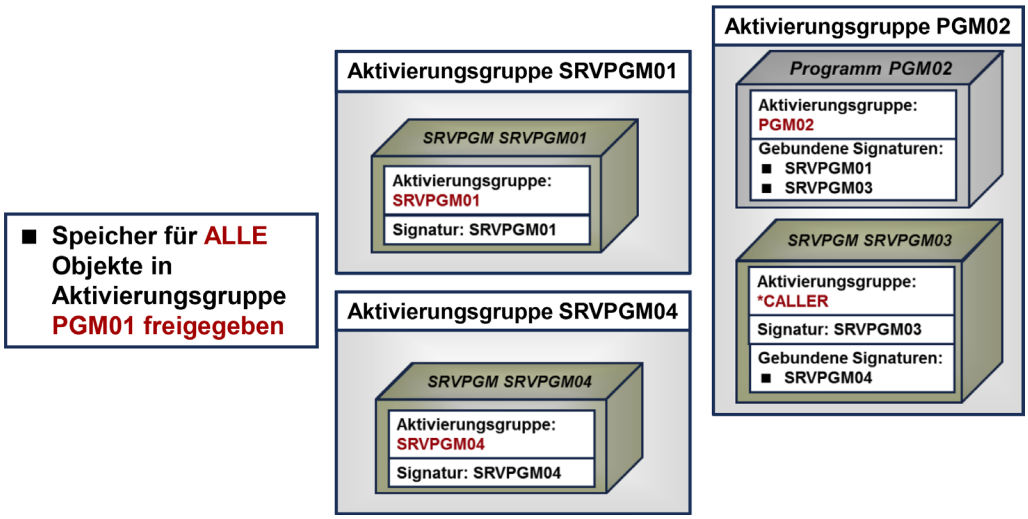


Aktivierung von Serviceprogrammen in benannten Aktivierungsgruppen

Die Serviceprogramme SRVPGM01 und SRVPGM04 werden jeweils in eigenen benannten Aktivierungsgruppen ausgeführt. Prozeduren in diesen Serviceprogrammen können aus allen innerhalb des Jobs aktivierten Programmen und Serviceprogrammen aufgerufen werden.

Wird die Aktivierungsgruppe PGM02 beendet, so bleiben die Serviceprogramme SRVPGM01 und SRVPGM04 weiterhin aktiviert. Gleiches gilt bei der Beendigung von Aktivierungsgruppe PGM01.

Das folgende Beispiel zeigt den Speicher, nachdem die Aktivierungsgruppe PGM01 beendet wurde. Lediglich Programm PGM01 und das in der gleichen Aktivierungsgruppe aktivierte Serviceprogramm SRVPGM02 wurden freigegeben.



Beendigung der benannten Aktivierungsgruppe PGM01

3.1.10.7.3 Aktivierungszeitpunkt von Serviceprogrammen

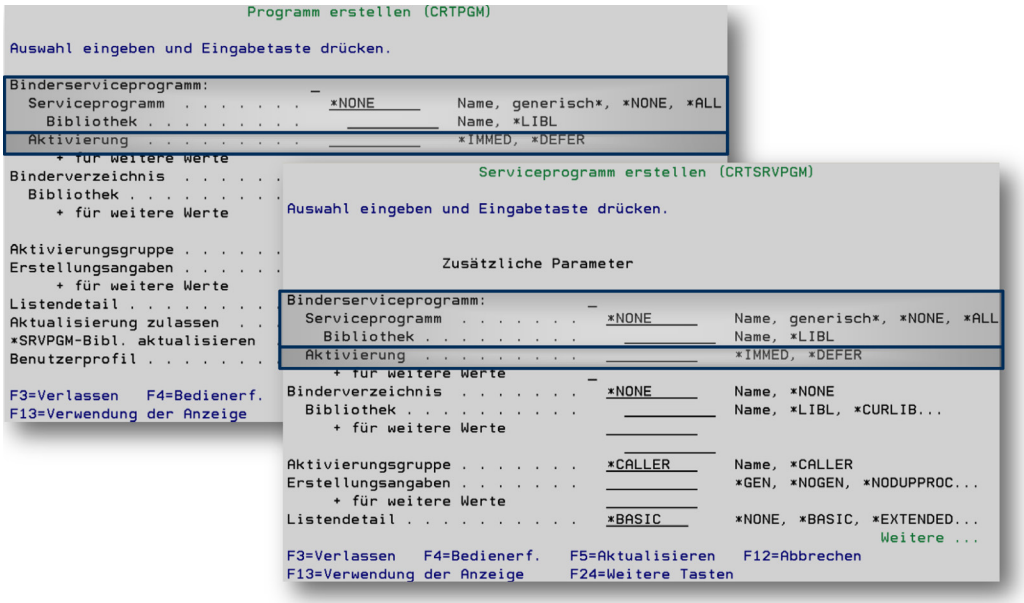
In hochmodularen Anwendungen kann es vorkommen, dass zu einem Programm oder Serviceprogramm viele weitere Serviceprogramme gebunden werden müssen. Je mehr Serviceprogramme gebunden werden, desto zeitaufwendiger die Aktivierung.

Es kann jedoch vorkommen, dass aus einigen der gebundenen Serviceprogramme während der Programmausführung auch während der gesamten Joblaufzeit keine Prozedur aufgerufen werden muss. Damit wurden Serviceprogramme unnötigerweise aktiviert.

Vor Release 6.1 wurden bei Programmaufruf immer alle zum Programm gebundenen Serviceprogramme aktiviert. Seit Release 6.1 ist es möglich, den Aktivierungszeitpunkt festzuglegen, das heißt, man kann angeben, ob die gebundenen Serviceprogramme direkt mit der Aktivierung des rufenden Programms aktiviert werden sollen oder erst beim Aufruf der ersten Prozedur aus dem Serviceprogramm.

Zu diesem Zweck wurden die Erstellungsbefehle CRTPGM (Programm erstellen) und CRTSRVPGM (Serviceprogramm erstellen) um die Option „Aktivierung“ erweitert, die in Verbindung mit Binder-Serviceprogrammen angegeben werden kann. Der Unterlassungswert für die Aktivierung von Binder-Serviceprogrammen ist weiterhin *IMMED, sprich die gebundenen Serviceprogramme werden sofort bei Aufruf des rufenden Programms aktiviert. Wird die Option jedoch auf *DEFER gesetzt, wird die Aktivierung bis zum Aufruf der ersten Prozedur aus dem Serviceprogramm verzögert.

Die folgende Abbildung zeigt die Option Binder-Serviceprogramm mit Aktivierung in den Erstellungsbefehlen CRTPGM und CRTSRVPGM:



Aktivierung von Binder-Serviceprogrammen

3.1.10
Seite 40

Sofern mit vielen unterschiedlichen Serviceprogrammen gearbeitet wird, bietet es sich an, diese in einem Binderverzeichnis zu hinterlegen. Dieses Binderverzeichnis wird anstatt der Binder-Serviceprogramme in den Erstellungsbefehlen CRTPGM, CRTSRVPGM und CRTBNDxxx angegeben. Im Binderschritt wird dann dieses Binderverzeichnis nach dem Serviceprogramm durchsucht, in dem sich die aufgerufenen Prozeduren befinden.

Beim Hinzufügen eines Serviceprogramms über den Befehl ADDBNDDIRE (Binderverzeichniseintrag hinzufügen), kann ebenfalls angegeben werden, ob dieses Serviceprogramm sofort oder erst bei Aufruf der ersten Prozedur aus dem Serviceprogramm aktiviert werden soll. Der Unterlassungswert für die Aktivierung ist auch bei diesem Befehl *IMMED.

Die folgende Abbildung zeigt den Befehl ADDBNDDIRE mit Objektspezifikationen und der Auswahl „Aktivierung“ für Serviceprogramme.

```
                Binderverz.eintrag hinzufügen (ADDBNDDIRE)

Auswahl eingeben und Eingabetaste drücken.

Binderverzeichnis  . . . . . __________    Name
  Bibliothek . . . . . . . .   *LIBL_____   Name, *LIBL, *CURLIB...
Objektspezifikationen:          _
  Objekt . . . . . . . . . . . __________   Name, generisch*, *ALL
    Bibliothek . . . . . . . .   *LIBL_____ Name, *LIBL
  Objektart  . . . . . . . . . *SRVPGM____  *SRVPGM, *MODULE
  Aktivierung  . . . . . . . . *IMMED_____  *IMMED, *DEFER
    + für weitere Werte         _
Positionsangaben:
  Objektposition . . . . . . . *LAST___     *LAST, *FIRST, *AFTER...
  Objekt . . . . . . . . . . . __________   Name
    Bibliothek . . . . . . . .   __________ Name, *LIBL
  Objektart  . . . . . . . . . _______      *SRVPGM, *MODULE



                                                                Ende
F3=Verlassen   F4=Bedienerf.   F5=Aktualisieren   F12=Abbrechen
F13=Verwendung der Anzeige     F24=Weitere Tasten
```

CL-Befehl ADDBNDDIRE – Aktivierung von Serviceprogrammen

Bei Ausführung des Befehls WRKBNDDIRE werden alle im Binderverzeichnis eingetragenen Serviceprogramme aufgelistet. Diese Anzeige enthält eine Spalte „Aktivierung“, in der für jedes erfasste Serviceprogramm angezeigt wird, ob es sofort oder verzögert aktiviert wird.

3.1.10.8 Aktivierungsgruppen und Dateiüberschreibungen

Sobald man die ersten Prozeduren in Serviceprogrammen hinterlegt und aus Programmen aufruft, können diese Programme nicht mehr mit der Default-Aktivierungsgruppe umgewandelt werden. Damit erhält man einen Mix aus OPM- und ILE-Programmen, die zumindest in zwei unterschiedlichen Aktivierungsgruppen ausgeführt werden, der Default-Aktivierungsgruppe und einer benannten Aktivierungsgruppe.

Auf den ersten Blick scheint diese Konstellation keine negativen Auswirkungen zu haben. OPM-Programme können ILE-Programme aufrufen und umgekehrt. Parameterwerte können zwischen ILE- und OPM-Programmen korrekt transferiert werden. Die Probleme treten erst auf den zweiten Blick auf, wenn zum Beispiel die Druckersteuerung, die auf OVRPRTF (Druckerdatei überschreiben) beruht, nicht mehr funktioniert.

Diese Probleme werden jedoch weder durch den Mix aus ILE- und OPM-Programmen noch durch die unterschiedlichen Aktivierungsgruppen verursacht. Schuld an der Misere sind die Unterlassungswerte, die in den Überschreibungs-(Override)-Befehlen bei den Optionen „Bereich überschreiben“ (OVRSCOPE) und „Offener Bereich“ (OPNSCOPE) hinterlegt wurden. Die Default-Werte beider Befehle wurden mit *ACTGRPDFN festgelegt, was bedeutet, dass die Überschreibung nur innerhalb der gleichen Aktivierungsgruppe gilt.

In der reinen OPM-Umgebung, in der alle Programme ohne wenn und aber in der Default-Aktivierungsgruppe ausgeführt werden, spielt es keine Rolle, ob die gesetzten Optionen auf Aktivierungsgruppenebene oder auf Jobebene Gültigkeit haben.

3.1.10
Seite 42

In gewachsenen Anwendungen, in denen Dateiüberschreibungen oft in einem separaten CL-Programm ausgeführt wurden, kann eine Überschreibung auf Aktivierungsgruppenebene zu massiven Problemen führen.

Beispiel:

Im Rahmen der Anwendungsmodernisierung wurden OPM-Programme überarbeitet und durch ILE-Programme ersetzt, die in einer benannten Aktivierungsgruppe ausgeführt werden.

An mehreren Stellen innerhalb der Anwendung wurden dazu Dateien zunächst in OPM-CL-Programmen überschrieben und im Anschluss mit RPG- oder Cobol-Programmen verarbeitet.

Solange es sich bei den Verarbeitungsprogrammen um OPM-Programme handelte, konnten die überschriebenen Dateien problemlos verarbeitet werden, auch ohne dass die Default-Werte geändert werden mussten. Sowohl die CL-Programme als auch die verarbeitenden Programme wurden in der Default-Aktivierungsgruppe ausgeführt, wodurch aus allen Programmen auf die überschriebenen Dateien zugegriffen werden konnte.

Die neuen ILE-Programme wurden mit einer benannten Aktivierungsgruppe erstellt. Da die Überschreibungsbefehle jedoch mit dem Unterlassungswert OVRSCOPE=*YES ausgeführt werden, gilt die Überschreibung nur innerhalb der Default-Aktivierungsgruppe. Die überschriebenen Dateien können somit nicht von den ILE-Programmen verarbeitet werden.

Treten bei Überschreibungen solche Probleme auf, sollte man jedoch keinesfalls die ILE-Programme mit Aktivierungsgruppe *CALLER erstellen. Damit wären zwar die Probleme mit der Dateiüberschreibung gelöst, die Probleme, die durch die Ausführung von ILE- und ILE-Serviceprogrammen in der Default-Aktivierungsgruppe auftreten können, sind jedoch weitaus schwerwiegender.

Die bessere Lösung ist, die Überschreibungsbefehle in den OPM-CL-Programmen dahingehend zu überarbeiten, dass die Überschreibung für alle Aktionen innerhalb des Jobs unabhängig von der Aktivierungsgruppe gilt (OVRSCOPE=*JOB).

Anmerkung:

Um Probleme mit Dateiüberschreibungen bei einem Mix aus OPM- und ILE-Programmen zu vermeiden, sollten die Überschreibungsbefehle nicht mit Default-Werten ausgeführt werden. Stattdessen sollte die Option OVRSCOPE (Bereich überschreiben) auf *JOB geändert werden, so dass die Überschreibung auf Jobebene gilt.

3.1.10.8.1 Dateiüberschreibungen setzen

In den folgenden Überschreibungsbefehlen wurde die Option „Bereich überschreiben“ (OVRSCOPE) per Default so gesetzt, dass die Überschreibung nur innerhalb der gleichen Aktivierungsgruppe gilt.

- OVRDBF: mit Datenbankdatei überschreiben
- OVRDKTF: mit Diskettendatei überschreiben
- OVRDSPF: mit Bildschirmdatei überschreiben
- OVRICFF: mit ICF-Datei überschreiben
- OVRPRTF: mit Druckerdatei überschreiben
- OVRSAVF: mit Sicherungsdatei überschreiben
- OVRTAPF: mit Banddatei überschreiben

Für jeden dieser Befehle kann die Option OVRSCOPE auf einen der folgenden Werte gesetzt werden:

- *ACTGRPDFN
  Bei dieser Ausprägung handelt es sich um den Unterlassungswert, der dafür sorgt, dass die Überschreibung nur innerhalb der Aktivierungsgruppe wirksam wird.
  Sofern die Überschreibung nicht explizit durch Ausführung des CL-Befehls DLTOVR (Überschreibung löschen) zurückgenommen wird, bleibt die Überschreibung bis zum Ende der Aktivierungsgruppe bestehen.
  Wird die Aktivierungsgruppe durch Ausführung des CL-Befehls RCLACTGRP (Aktivierungsgruppe freigeben) beendet, werden auch die zur Aktivierungsgruppe gehörenden Überschreibungen automatisch gelöscht.
- *JOB
  Wird ein Überschreibungsbefehl mit der Option OVRSCOPE=*JOB ausgeführt, erfolgt die Überschreibung auf Jobebene, das heißt, die Überschreibung gilt für alle Aktionen, die innerhalb des Jobs ausgeführt werden. Dabei spielt es keine Rolle, aus welcher Aktivierungsgruppe die Aktion initiiert wurde.
  Die Überschreibung gilt so lange, bis sie durch Ausführung eines entsprechenden DLTOVR-Befehls zurückgenommen wird.
  Eine Überschreibung mit der Option OVRSCOPE=*JOB kommt der Handhabung von Überschreibungen in reinen OPM-Umgebungen am nächsten.

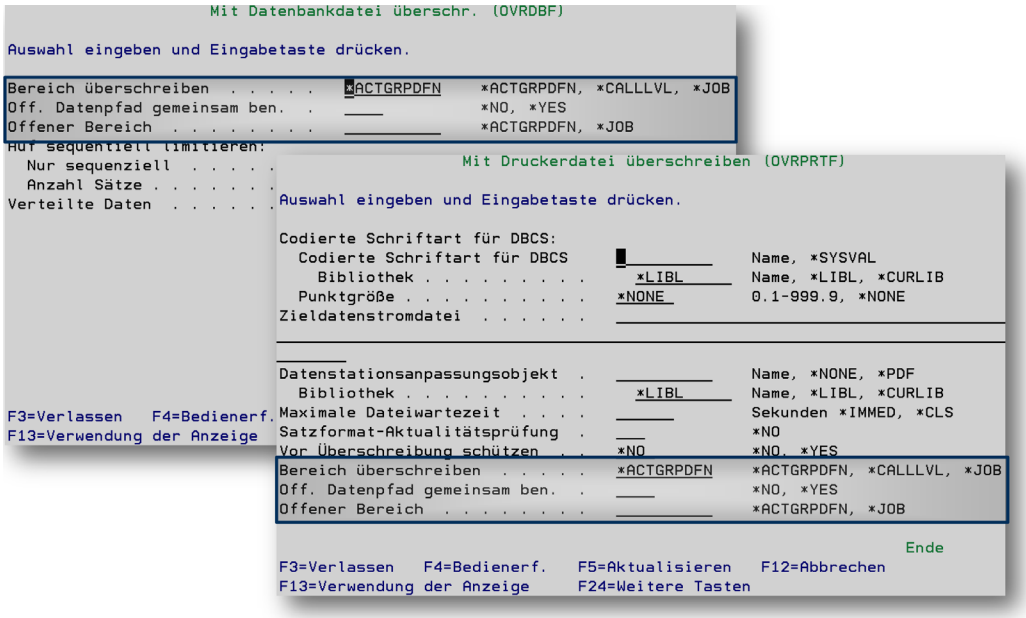
- *CALLLVL
  Sofern die Überschreibung mit Option OVRSCOPE=*CALLLVL erfolgt, gilt die Überschreibung für alle nachfolgend im gleichen Call-Stack aufgerufenen Programme und Prozeduren. Dabei spielt es keine Rolle, mit welcher Aktivierungsgruppe das Programm oder das Serviceprogramm, in der die Prozedur hinterlegt ist, erstellt wurde.

Es besteht jedoch auch die Möglichkeit, eine Überschreibung durch die Ausführung eines entsprechenden DLTOVR-Befehls zurückzunehmen.

Die Überschreibung mit Option OVRSCOPE=*CALLLVL hat für Programme und Prozeduren, die nicht im Call-Stack auf das Programm, das die Überschreibung ausgeführt hat, folgen, keine Auswirkungen.

Der überschriebene Bereich (OVRSCOPE) ist nicht die einzige Option, die in den Überschreibungsbefehlen wahlweise für die Aktivierungsgruppenebene oder das Joblevel angegeben werden kann. So kann auch der offene Bereich (Option OPNSCOPE) wahlweise mit der Ausprägung *ACTGRPDFN oder *JOB angegeben werden.

Die folgende Abbildung zeigt die beiden Überschreibungsbefehle OVRDBF (Mit Datenbank überschreiben) und OVRPRTF (Mit Druckerdatei überschreiben) mit den Optionen „Bereich überschreiben“ (OVRSCOPE) und „Offener Bereich“ (OPNSCOPE).



Überschreibungsbefehle OVRDBF und OVRPRTF mit Optionen OVRSCOPE und OPNSCOPE

Die Frage, die sich nun stellt, ist die nach der besten Überschreibungsoption. Die Antwort ist – wie so oft: je nachdem.

In über die Jahre gewachsenen Anwendungen wurde vor Aufruf des eigentlichen RPG- oder Cobol-Programms zunächst oft ein CL-Programm ausgeführt, das die zu verarbeitenden Dateien überschrieb. Sei es, um auf eine spezielle Teildatei zugreifen zu können oder um eine bestimmte Datei verarbeiten zu können, ohne dass die Bibliotheksliste geändert werden musste.

Um mit gewachsenen Anwendungen nicht in Konflikt zu geraten und trotzdem ILE- und Serviceprogramme in unterschiedlichen Aktivierungsgruppen ausführen zu können, die die überschriebenen Dateien verarbeiten müssen, sollten im ersten Schritt der Modernisierung die Überschreibungen auf Jobebene (OVRSCOPE=*JOB) erfolgen.

Anmerkung:

In gewachsenen OPM-Anwendungen sollten Überschreibungen immer mit der Option OVRSCOPE=*JOB erfolgen.

Bei neuen Programmen – oder auch bei Modernisierung einer vorhandenen Anwendung – sollte man jedoch versuchen, Dateiüberschreibungen weitgehend zu vermeiden. Gerade im Bereich der Datenbanken gibt es heute Alternativen, durch die sehr flexibel auf Dateien/Tabellen, Teildateien und Views zugegriffen werden kann, ohne dass eine Überschreibung notwendig wäre.

So können in RPG Datei, Bibliothek und/oder Teildatei zur Laufzeit in den F-Bestimmungen mit Hilfe der Schlüsselworte EXTFILE und/oder EXTMBR gesetzt werden. Dabei können die Datei, Bibliothek und/oder Teildatei entweder fix vorgegeben oder über Variablen gesetzt werden. Die einzige Voraussetzung für diese Art der Verarbeitung ist, dass die Dateien erst nach dem Initialisieren der Variablen geöffnet werden dürfen. Diese Möglichkeit gibt es bereits seit Release V5R1M0.

Beim Einsatz von RPGOA (Open Access) kann mit Hilfe von entsprechenden Handler-Prozeduren auf beliebige Dateien unterschiedlichster Art (unter anderem auch auf HTML- oder PDF-Dokumente) zugegriffen werden.

Obwohl es möglich ist, die überschriebenen Dateien mit embedded SQL zu verarbeiten, ist das nicht zwingend notwendig. So kann anstelle der überschriebenen Datei ein Alias verwendet werden. Unter einem Alias versteht man ein permanentes Objekt, das lediglich einen Verweis auf eine Tabelle, physische Datei, Teildatei oder View darstellt. Aliase können mit SQL wie jede beliebige Tabelle oder View verarbeitet werden.

Eine andere Alternative ist, dynamisches SQL zu verwenden. Dabei wird das auszuführende SQL-Statement erst zur Laufzeit als Text aufbereitet. Dieser Text wird durch die Ausführung des SQL-Statements PREPARE in ein ausführbares SQL-Statement konvertiert und kann im Anschluss ausgeführt werden.

Auch wenn Überschreibungen im Bereich von Datenbankdateien weitgehend vermieden werden können, so werden sie in anderen Bereichen – etwa in Verbindung mit der Druckersteuerung – auch heute noch rege genutzt. So kann zur Laufzeit mit Hilfe des OVRPRTF-Befehls die Ausgabewarteschlange flexibel zugewiesen werden. Ebenso kann die Anzahl der Kopien variabel festgelegt werden. Des Weiteren ist es auch möglich, wahlweise eine Spoolfile oder ein PDF-Dokument zu erzeugen.

Sofern Dateiüberschreibungen auch bei neuen Programmen und Anwendungen erforderlich sind, und weiterhin ein Mix aus OPM und ILE-Programmen in unterschiedlichen Aktivierungsgruppen besteht, sollten die Überschreibungen mit dem OVRSCOPR *CALLVL erfolgen. Die Überschreibung gilt nur für den aktuellen Call-Stack, jedoch unabhängig davon, in welcher Aktivierungsgruppe die aufgerufenen Programme und Prozeduren ausgeführt werden.

Anmerkung:

Sofern Überschreibungen in neuen (ILE-)Programmen oder Anwendungen erforderlich sind, sollten sie mit der Option OVRSCOPE=*CALLLVL erfolgen.

3.1.10.8.2 Dateiüberschreibungen zurücknehmen

Dateiüberschreibungen werden mit den Überschreibungsbefehlen auf Joblevel, Aktivierungsgruppenebene oder für den aktuellen Call-Stack gesetzt.

Die Überschreibungen werden – abhängig von der Art der Überschreibung – beim Jobende, beim Ende der Aktivierungsgruppe oder beim Beenden des initiierenden Programms im Call-Stack automatisch zurückgenommen. Es gibt jedoch Situationen, in denen das nicht ausreicht, sprich in denen die Überschreibung manuell zurückgesetzt werden muss.

Über den Befehl DLTOVR (Überschreibung löschen) können Überschreibungen zurückgesetzt werden. Obwohl es abhängig vom Objekttyp unterschiedliche Überschreibungsbefehle (etwa OVRDBF, OVRPRTF) gibt, gibt es nur einen einzigen Befehl, über den die Überschreibungen zurückgenommen werden können.

Über die Option „Aufrufebene“ (LVL) kann die Art der Überschreibung, die zurückgenommen werden soll, festgelegt werden. Für die Aufrufebene (Option LVL) können die folgenden Ausprägungen angegeben werden.

- *ACTGPDFN Unterlassungswert
  Überschreibungen mit Überschreibungsbereich „Aktivierungsgruppe“ (Option OVRSCOPE im Überschreibungsbefehl = *ACTRPDFN), werden zurückgesetzt.
  Überschreibungen mit einem anderen Überschreibungsbereich bleiben unangetastet.
- *JOB
  Überschreibungen auf Jobebene (Option OVRSCOPE im Überschreibungsbefehl = *JOB) werden zurückgenommen.
- *
  Überschreibungen für den Call-Stack (Option OVRSCPOPE im Überschreibungsbefehl = *CALLVL) werden zurückgenommen.

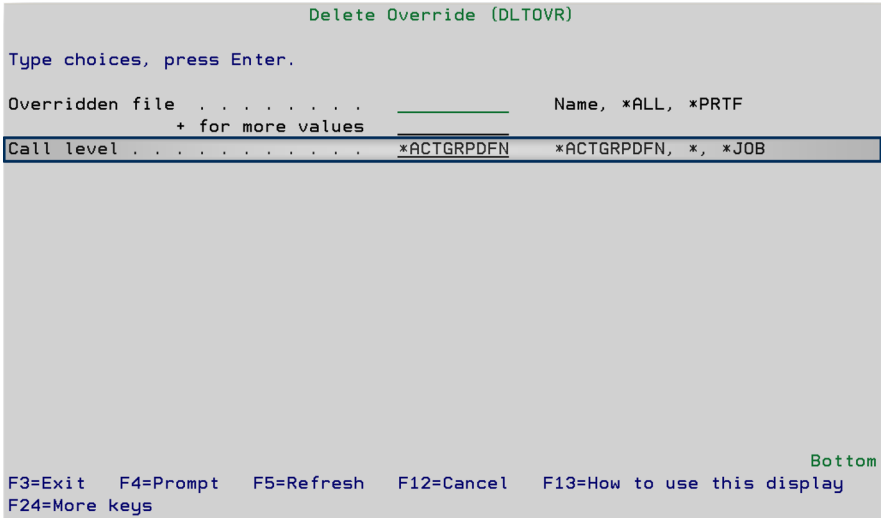
Achtung:

Die Ausprägungen für den Call-Level sind in der Option OVRSCOPE in den Überschreibungsbefehlen und in der Option LVL im Befehl DLTOVR unterschiedlich!

OVRSCOPE = *CALLVL und LVL = *

3.1.10
Seite 48

Die folgende Abbildung zeigt den Befehl CL DLTOVR (Überschreibung löschen), über den wahlweise einzelne oder alle Überschreibungen oder nur Überschreibungen für Printerfiles gelöscht werden können. Über den zweiten Parameter wird gesteuert, ob die Überschreibungen, die auf Jobebene gesetzt wurden, oder diejenigen, die innerhalb der Aktivierungsgruppe gültig sind oder die für den aktuellen Call-Stack erfolgten, gelöscht werden.



CL-Befehl DLTOVR = Überschreibungen löschen

3.1.10.9 Aktivierungsgruppen und Commit-Definitionsbereich

Arbeitet man unter Commitment Control, kann man bei der Umstellung von reinen OPM-Umgebungen auf Umgebungen aus einem Mix an OPM- und ILE-Programmen sowie in reinen ILE-Umgebungen ähnliche Überraschungen erleben wie bei den Überschreibungsbefehlen.

Auch hier verursacht ein Default-Wert in dem Befehl STRCMTCTL (Commit-Steuerung starten) das Problem.

Der Unterlassungswert für die Option Commit-Definitionsbereich (CMTSCOPE) im Befehl STRCMTCTL wurde per Default auf *ACTGRP gesetzt. Das bedeutet, dass Änderungen nur innerhalb der gleichen Aktivierungsgruppe festgeschrieben oder zurückgesetzt werden.

Die folgende Abbildung zeigt den CL-Befehl STRCMTCTL (Commit-Steuerung starten) mit der Default-Ausprägung *ACTGRP der Option COMMIT-Definitionsbereich.

```
                    COMMIT-Steuerung starten (STRCMTCTL)

Auswahl eingeben und Eingabetaste drücken.

Sperrebene . . . . . .   _           *CHG, *CS, *ALL
Mitteilungsobjekt:
  Objekt . . . . . . .   *NONE       Name, *NONE
    Bibliothek . . . .     _____     Name, *LIBL, *CURLIB
  Objektart:
  (*MSGQ *DTAARA oder *FILE) . .     *MSGQ, *DTAARA, *FILE
  Teildatei, wenn *FILE . . . .      Name, *FIRST
COMMIT-Definitionsbereich . . .  *ACTGRP       *ACTGRP, *JOB
Text 'Beschreibung' . . . . . .  *DFTTEXT
_____
Journal . . . . . .   *NONE       Name, *NONE
  Bibliothek . . . .     _____     Name, *LIBL, *CURLIB
Wegzulassende Journaleinträge .  *NONE         *NONE, *LUWID



                                                              Ende
F3=Verlassen   F4=Bedienerf.   F5=Aktualisieren   F12=Abbrechen
F13=Verwendung der Anzeige     F24=Weitere Tasten
```

CL-Befehl STRCMTCTL – Commit-Steuerung starten

In ILE-Umgebungen – insbesondere, wenn mit mehreren Aktivierungsgruppen gearbeitet wird – kann es jedoch vorkommen, dass sich eine Transaktion über mehrere Aktivierungsgruppen erstreckt und entweder als Einheit abgeschlossen oder komplett zurückgesetzt werden muss. In solchen Fällen sollte der Job als Commit-Definitionsbereich verwendet werden.

3.1.10
Seite 50

Leider kann die Commitment-Steuerung innerhalb eines Jobs nur einmalig gestartet werden. Ist die Commitment-Steuerung erst einmal gestartet, werden alle Transaktionen – abhängig vom Commit-Definitionsbereich – nach den gleichen Regeln festgeschrieben bzw. zurückgesetzt.

Sofern die Commitment-Steuerung nach anderen Regeln ausgeführt werden soll, sprich mit anderen Optionen, muss sie zunächst durch Ausführung des CL-Befehls ENDCMTCTL (Commitment-Steuerung beenden) beendet werden. Erst dann kann der Befehl STRCMTCTL (geänderte Optionen) erneut ausgeführt werden.

Sofern beim Beenden der Commitment-Steuerung noch nicht alle Transaktionen abgeschlossen sind, wird in interaktiven Jobs eine entsprechende Fehlermeldung ausgegeben, die beantwortet werden muss. In Batch-Verarbeitungen werden die nicht festgeschriebenen Änderungen rückgängig gemacht. Aus diesem Grund ist das Beenden und erneute Starten der Commitment-Steuerung mit Vorsicht zu behandeln.

Anmerkung:

Um Problemen mit Commit und Rollback in gemischten OPM- und ILE-Anwendungen aus dem Weg zu gehen, sollte die Commitment-Steuerung im Job-Startprogramm direkt zu Beginn des Jobs mit CMTSCOPE=*JOB gestartet werden.

3.1.10.9.1 Commitment-Steuerung und SQL

Werden in einem reinen RPG-Programm Datensätze unter Commitment-Steuerung hinzugefügt, geändert oder gelöscht (Schlüsselwort COMMIT in den F-Bestimmungen), muss der Befehl STRCMTCTL (Commitment-Steuerung starten) vorab explizit im aktuellen Job ausgeführt worden sein. Sofern die Commitment-Steuerung nicht gestartet, sprich der Befehl STRCMTCTL noch nicht ausgeführt wurde, wird das Programm sofort bei Aufruf mit einer Fehlermeldung abgebrochen.

Werden in einem Programm Datensätze mit (embedded) SQL geändert, hinzugefügt oder gelöscht, wird beim Erstellen des Programms oder Moduls angegeben, ob diese Aktionen unter Commitment-Steuerung ausgeführt werden oder nicht. Ob die Datensätze unter Commiment-Steuerung verarbeitet werden und welches Commitment-Level dabei verwendet werden soll, wird über die Option COMMIT in den Kompilierungsbefehlen (zum Beispiel CRTSQLRPGI) angegeben. Der Unterlassungswert für diese Option ist *CHG, das heißt, es wird davon ausgegangen, dass die SQL-Data-Manipulation-Befehle unter Commimtent-Steuerung verarbeitet werden. Sofern ohne Commitment-Control gearbeitet wird, muss die Compile-Option COMMIT explizit auf *NONE gesetzt werden. Alternativ kann auch ein entsprechendes SET OPTION-Statement im Quellcode hinterlegt werden.

Wurde vor Aufruf eines embedded SQL-Programms, das unter Commitment-Steuerung ausgeführt wird, der Befehl STRCMTCTL innerhalb des Jobs noch nicht explizit aufgerufen, bricht das Programm im Gegensatz zu einem reinen RPG-Programm, das unter Commitment-Steuerung ausgeführt wird, nicht ab. Stattdessen startet SQL die Commitment-Steuerung eigenständig, das heißt, der Befehl STRCMTCTL (Commitment-Steuerung starten) wird beim Programmstart automatisch mit den Default-Einstellungen ausgeführt.

Da die Option Commit-Bereich (CMTSCOPE) im Befehl STRCMTCTL per Default auf Aktivierungsgruppenebene (*ACTGRP) gesetzt ist, kann das automatische Starten der Commitment-Control insbesondere in Umgebungen, in denen sich Transaktionen über mehrere Aktivierungsgruppen erstrecken können, zu bösen Überraschungen führen. So kann es sein, dass bei Commit die Änderungen nur teilweise festgeschrieben und beim Rollback nur unvollständig zurückgesetzt werden.

Anmerkung:

Um zu verhindern, dass SQL im Bedarfsfall die Commitment-Steuerung selbständig startet, sollte die Commitment-Steuerung jeweils zu Beginn eines Jobs mit den gewünschten Optionen gestartet werden.

3.1.10.10 Aktivierungsgruppen und Open Data Paths (ODPs)

Eine weitere Besonderheit, die man bei der Verwendung von embedded SQL beachten sollte, sind die offenen Datenpfade (ODP = Open Data Path).

Jedes SQL-Statement, das ausgeführt wird, muss zur Laufzeit optimiert werden. Das heißt, der beste Zugriffsweg muss ermittelt werden, temporäre Objekte, die zur Ausführung der Abfrage erforderlich sind, müssen erstellt, miteinander verlinkt und mit Daten gefüllt werden. Erst im Anschluss können die ausgewählten Daten (jm Programm) verarbeitet werden. Der zeitaufwendigste Part während des FULL OPEN-Prozesses ist die Öffnung des Datenpfads, sprich das Erstellen der temporären Objekte.

Um die Optimierungszeit so kurz wie möglich zu halten, wird versucht, den ODP nicht zu schließen, also die temporären Objekte nicht zu löschen. Bei der nächsten Ausführung des gleichen SQL-Statements müssen lediglich die Daten in den temporären Objekten aktualisiert werden. Diesen Prozess nennt man PSEUDO OPEN, im Gegensatz zum FULL OPEN, bei dem die komplette Optimierung erfolgen muss. Die Ausführungszeit bei der wiederholten Ausführung des gleichen SQL-Statements innerhalb des gleichen Job kann durch PSEUDO OPEN zum Teil auf einen Bruchteil der FULL OPEN-Zeit reduziert werden. Voraussetzung für einen PSEUDO OPEN ist, dass das SQL-Statement zum einen wiederverwendbar ist und dass der ODP nicht geschlossen/gelöscht wird.

Ob und wann der ODP explizit geschlossen wird, wird im Kompilierungsbefehl (etwa CRTSQLRPGI oder CRTSQLCBLI) durch die Option „SQL Cursor schließen“ (CLOSQLCSR) gesteuert. Die folgenden Optionen können ausgewählt werden:

- *ENDACTGRP — Die offenen Datenpfade (ODP) werden bei wiederverwendbaren SQL-Statements bis zum Ende der Aktivierungsgruppe offengehalten.
  Bei Folgeaufrufen kann ein PSEUDO OPEN erfolgen, das heißt, es werden lediglich die Daten in den ODPs aktualisiert.
  Die Option *ENDACTGPR ist als Unterlassungswert in den Compile-Befehlen hinterlegt.

• *ENDMOD	Alle innerhalb des Moduls geöffneten Datenpfade (ODPs) werden beim Verlassen/ Beenden des Moduls gelöscht. Da die ODPs beim Verlassen des Moduls gelöscht wurden, muss bei einem erneuten Aufruf wieder ein FULL OPEN erfolgen, also eine komplette Optimierung.

Anmerkung:

Das SQL-Statement CLOSE, das im Quellcode hinterlegt werden kann, schließt lediglich einen Cursor, löscht jedoch nicht den ODP.

Um die (embedded) SQL-Statements so performant wie möglich ausführen zu können, sollte versucht werden, die Open Data Paths (ODPs) solange wie möglich geöffnet zu halten. Leider gibt es für die Kompilierungsoption CLOSQLCSR in ILE-Programmen keine Ausprägung, die es ermöglichen würde, die ODPs auf Jobebene offen zu halten. Aus diesem Grund sollte man die Option CLOSQLCSR=*ENDACTGRP im Kompilierungsbefehl wählen.

Werden die Programme, in denen Module mit embedded SQL gebunden sind, mit Aktivierungsgruppe *NEW erstellt, können die ODPs ebenfalls nicht offengehalten werden. Bei jedem Aufruf wird eine neue Aktivierungsgruppe – in der auch die ODPs aktiviert werden – gebildet und beim Beenden wieder komplett (inklusive der ODPs) gelöscht. Beim erneuten Aufruf des Programms muss für jedes SQL-Statement erneut ein FULL OPEN ausgeführt werden.

Anmerkung:

Programme mit embedded SQL sollten nur in Ausnahmefällen mit der Aktivierungsgruppe *NEW erstellt werden. Besser ist es, ILE-Programme mit embedded SQL in benannten Aktivierungsgruppen auszuführen. Benannte Aktivierungsgruppen ermöglichen es, die Anzahl der FULL OPENS auf ein Minimum zu reduzieren. Zum anderen können benannte Aktivierungsgruppen jederzeit mit dem CL-Befehl RCLACTGRP (Aktivierungsgruppe zurückfordern) beendet werden.

3.1.10.10.1 Empfehlungen im Umgang mit Aktivierungsgruppen

Zum Abschluss noch einige Tipps und Empfehlungen zur Planung und Verwendung von Aktivierungsgruppen

Benannte Aktivierungsgruppen für (ILE-)Programme
Werden ILE-Programme in benannten Aktivierungsgruppen (zum Beispiel Aktivierungsgruppenname = Programmname) ausgeführt, bleibt der Speicher reserviert, so dass bei Folgeaufrufen nur ein minimaler Aktivierungsoverhead entsteht.

Bei ILE-Programmen mit embedded SQL bleiben die ODPs geöffnet, sofern sie wiederverwendbar sind und die Compile-Option CLOSQLCSR=*ENDACTGRP gesetzt wurde. Bei Folgeaufrufen ist keine volle Optimierung erforderlich, stattdessen kann ein PSEUDO OPEN erfolgen, das heißt, lediglich die Daten müssen aktualisiert werden.

Die aktivierten Programme können jederzeit innerhalb des Jobs mit Hilfe des CL-Befehls RCLACTGRP (Aktivierungsgruppe zurückfordern) explizit geschlossen werden.

Aktivierungsgruppe *NEW für Programme, die nur einmalig ausgeführt werden
Programme, die nur einmalig – etwa zum Monatsabschluss oder nachts – direkt nach der Sicherung ausgeführt werden, können mit der Aktivierungsgruppe *NEW erstellt werden.

ILE-Programme, die rekursiv ausgeführt werden, sollten nicht länger mit Aktivierungsgruppe *NEW erstellt und ausgeführt werden. Stattdessen sollte die zyklische Main-Procedure, die die Rekursion verhindert, durch eine lineare Main-Procedure ersetzt werden.

Benannte Aktivierungsgruppe für Serviceprogramme mit universell einsetzbaren Prozeduren
Beim Erstellen von Serviceprogrammen wird für die Aktivierungsgruppe der Unterlassungswert *CALLER angegeben. Serviceprogramme, die mit Aktivierungsgruppe *CALLER erstellt wurden, werden in der gleichen Aktivierungsgruppe wie die rufenden Programme aktiviert.

Bei Serviceprogrammen, die universell einsetzbare Prozeduren (etwa Datumsfunktionen) enthalten, genügt es, dass das Serviceprogramm einmal pro Job und nicht für jedes Programm aktiviert wird. Solche Serviceprogramme sollten in einer benannten Aktivierungsgruppe (zum Beispiel Serviceprogrammname = Aktivierungsgruppenname) erstellt werden.

Aktivierungsgruppe *CALLER für Serviceprogramme mit Insert/Update oder Delete-Prozeduren
Serviceprogramme mit Prozeduren, in denen Datensätze hinzugefügt, geändert oder gelöscht werden, sollten mit Aktivierungsgruppe *CALLER erstellt werden.

Zum einen werden dadurch die Serviceprogramme zusammen mit den rufenden Programmen als Einheit ausgeführt, zum anderen können dadurch auch Transaktionen auf eine Aktivierungsgruppe konzentriert werden. Letzteres ist insbesondere dann wichtig, wenn mit Commitment-Steuerung gearbeitet wird. Der Unterlassungswert für den Commit-Definitionsbereich im Befehl Commit-Steuerung starten (STRCMTCTL) ist *ACTGRP, das heißt, Festschreiben von Änderungen bzw. das Zurücksetzen erfolgt nur innerhalb der Aktivierungsgruppe.

Aktivierung von Serviceprogrammen verzögern
Programme werden bei Aufruf aktiviert. Serviceprogramme werden per Default zusammen mit dem rufenden Programm aktiviert.

In hochmodularen Anwendungen kann es vorkommen, dass Serviceprogramme aktiviert werden, weil in Ausnahmefällen spezielle Prozeduren aus diesen Serviceprogrammen ausgeführt werden. Vielfach ist es jedoch so, dass bis zum Jobende diese Ausnahmefälle nicht eintreten und die Serviceprogramme daher umsonst aktiviert wurden.

Seit Release 6.1 besteht die Möglichkeit, die Aktivierung von Serviceprogrammen so lange hinauszuzögern, bis die erste Prozedur aus dem Serviceprogramm aufgerufen wird. Diese Option sollte man für (alle) Serviceprogramme in Betracht ziehen.

Commitment-Steuerung auf Jobebene starten
Arbeitet man mit Commitment-Steuerung, sollte man bereits zu Beginn eines Jobs den Befehl STRCMTCTL ausführen. Dabei sollte man den Commit-Definitionsbereich auf *JOB ändern.

Die Verwendung des Unterlassungswerts *ACTGRP ist nur dann zu empfehlen, wenn sichergestellt ist, dass Transaktionen sich nie über mehrere Aktivierungsgruppen erstrecken.

Überschreibungen innerhalb des Call-Stacks
Sofern es für Dateiüberschreibungen Alternativen gibt (etwa Schlüsselworte in den F-Bestimmungen oder embedded SQL), sollten diese genutzt werden.

Sofern man sich in einem Mix aus OPM- und ILE-Programmen weiterhin mit Dateiüberschreibungen auseinandersetzten muss, sollte man den Unterlassungswert für den Überschreibungsbereich in den CL-Befehlen von *ACTGRPDFN auf *JOB ändern. Dadurch gilt die Dateiüberschreibung nicht nur in der aktuellen Aktivierungsgruppe, sondern für alle Aktionen innerhalb des Jobs.

3.1.10
Seite 56

Sofern auch für neue Anwendungen und Programme – beispielsweise im Bereich der Druckersteuerung – Dateiüberschreibungen erforderlich sind, sollte der Überschreibungsbereich in den Überschreibungsbefehlen auf *CALLLVL geändert werden. Damit gilt die Überschreibung nur innerhalb des Call-Stacks für alle nachfolgend aufgerufenen Programme und Prozeduren.

3.1.11 Tipps für ILE-Anwendungen

Zum Abschluss noch einige Tipps zur erfolgreichen Implementierung der ILE-Konzepte:

Prozeduren sollten gekapselt werden
Vielfach werden Prozeduren als verbesserte Sub-Routinen angesehen.

An Prozeduren können im Gegensatz zu Sub-Routinen Parameter übergeben werden. Des Weiteren ist es möglich, innerhalb von Prozeduren lokale Variablen zu definieren und zu verwenden.

Wie aus Sub-Routinen kann auch aus Prozeduren auf globale Variablen zugegriffen werden.

Im Hinblick auf zukünftige Weiterentwicklungen sollten man Prozeduren kapseln (unabhängig davon, ob diese nur intern verwendet oder universell eingesetzt werden können), das heißt, der Datenaustausch zwischen der rufenden und der ausgeführten Prozedur sollte nur über Parameter erfolgen.

Innerhalb von Prozeduren sollte nur in Ausnahmefällen auf globale Variablen zugegriffen werden. Wesentlich besser ist es, die benötigten Informationen über Parameter auszutauschen.

Exportierte Prozeduren wann immer möglich
Sauber gekapselte Prozeduren können als „Mini-Programme“ angesehen werden.

Anstatt sich bei der Erstellung einer Prozedur zu überlegen, ob diese nur einmalig verwendet werden kann oder ob auch andere Programme und Prozeduren davon profitieren können, sollte man die Prozeduren von vornherein so gestalten, dass sie jederzeit von jeder anderen Prozedur (in einem anderen (Service-)Programm verwendet werden können.

Jedes kleine Stück Quellcode, das mehrfach codiert wird, sollte als exportierte Prozedur erstellt werden. In manchen Fällen handelt es sich dabei sogar nur um ein einziges Statement (etwa wenn mehrere Built-In-Funktionen ineinander geschachtelt werden müssen).

Prozeduren sollten nach Funktionalität in Teildateien zusammengruppiert werden
Prozeduren sollten nach Funktionalität in Teildateien zusammengruppiert werden. So können zum Beispiel alle Datumsfunktionen in einer Teildatei angeordnet werden. In einer weiteren Teildatei finden sich alle String-Funktionen. Alle Prozeduren, in denen auf eine bestimmte Tabelle/physische Datei zugegriffen wird, befinden sich in einer weiteren Teildatei.

Neue Prozeduren sollten immer am Ende des Quellcodes hinzugefügt werden. Zum einen können dadurch die Kollegen neue Funktionen einfacher lokalisieren und zum anderen kann auch die Binderquelle leichter aktualisiert werden, da die Prozeduren bereits in der richtigen Reihenfolge in der Quelle stehen.

Namenskonventionen sind immens wichtig
In hochmodularen Umgebungen wird es eine Vielzahl von Prozeduren geben, die wiederum in einer Vielzahl von Teildateien angeordnet sind.

Die Entwickler müssen vorhandene Prozeduren zum einen sehr schnell finden und zum anderen fehlende Prozeduren an der richtigen Stelle hinzufügen können, so dass der Kollege die neue Prozedur im Bedarfsfall ebenfalls schnell finden und einsetzen kann.

Grundlage hierfür sind strikte Namenskonventionen.

Die Namenskonventionen sollten sogar vor dem Erstellen der ersten ILE-Prozedur bestens durchdacht, dokumentiert und abgesegnet sein.

Da Namen von Variablen, Funktionen und Prozeduren bis zu 4.096 Zeichen lang sein können, sollte man sprechende Namen verwenden. Die Zeiten, in denen man mit 6 oder 8 Zeichen auskommen musste, sind vorbei. Zu kurze Namen sind genauso unvorteilhaft wie zu lange Namen.

Mehrere Prozeduren pro Modul, jedoch ein Serviceprogramm/ Programm pro Modul
Sofern die einzelnen Prozeduren bereits nach Funktionalität zusammengruppiert wurden, bietet es sich an, aus jeder Teildatei zunächst ein Modul und im Anschluss daran ein Serviceprogramm mit dem gleichen Namen zu erstellen.

Binderquelle für die exportierten Prozeduren
Für jedes Serviceprogramm sollte eine Binderquelle erstellt werden, in der die exportierten Prozeduren aufgelistet werden.

Neue Prozeduren müssen dabei immer am Ende der Liste hinzugefügt werden.

Sofern die neuen Funktionen immer am Ende des Quellcodes eingefügt werden, ist es ein Leichtes, die neuen Prozeduren zu lokalisieren und in die Binderquelle einzutragen.

Ein Binderverzeichnis mit allen Serviceprogrammen
Erstellt man aus jeder Teildatei, in der viele exportierte Prozeduren hinterlegt sind, ein Serviceprogramm, so ist man sehr flexibel, was den Aufruf von Prozeduren und die Aktivierung von Serviceprogrammen angeht. So kann man aus einem einzigen Programm Prozeduren aus einer Vielzahl an Serviceprogrammen aufrufen. Diese Prozeduren wiederum können weitere Prozeduren in anderen Serviceprogrammen aufrufen.

Um die Erstellung von Programmen und Serviceprogrammen so einfach wie möglich zu gestalten, sollte man alle Serviceprogramme in ein einziges Binderverzeichnis eintragen. In manchen Situationen bietet es sich auch an, ein Binderverzeichnis pro Bibliothek zu verwenden. Wichtig ist auf alle Fälle, dass ein Serviceprogramm nur in einem einzigen Binderverzeichnis eingetragen wird und die Prozedur/der Funktionsname eindeutig sind.

Beim Eintrag der Serviceprogramme in die Binderverzeichnisse sollte man dafür sorgen, dass die Aktivierung der Serviceprogramme bis zum Aufruf der ersten Prozedur aus dem entsprechenden Serviceprogramm verzögert wird. Dadurch werden nur die aktuell benötigten Serviceprogramme aktiviert.

Beim Binden von Programmen oder Serviceprogrammen werden lediglich das oder die Binderverzeichnisse angegeben, in denen die Serviceprogramme eingetragen sind. Damit können auch in hochmodularen Anwendungen mit einer Vielzahl von Serviceprogrammen und noch mehr exportierten Prozeduren und Funktionen die verwendeten Serviceprogramme einfach lokalisiert und gebunden werden.

Quellcode für Programme besteht nur aus Main-Procedure
Ein Programm sollte nur aus einem einzigen Modul bestehen.

Der Quellcode für ein Programm sollte im Idealfall nur noch aus einer Main-Procedure und ggf. einigen wenigen internen Prozeduren bestehen.

Die übrige Funktionalität sollte in exportierten Prozeduren in Serviceprogrammen hinterlegt sein, die aus dem Programmcode aufgerufen werden.

Bei neuen Programmen sollte man sich gegen eine zyklische und für eine lineare Main-Procedure entscheiden.

Programme in benannten Aktivierungsgruppen
Programme sollten in benannten Aktivierungsgruppen ausgeführt werden, zum Beispiel Aktivierungsgruppenname = Programmname.

Werden Programme nur einmalig aufgerufen, kann anstatt einer benannten Aktivierungsgruppe auch Aktivierungsgruppe *NEW verwendet werden.

Serviceprogramme mit allgemein gültigen Prozeduren in benannten Aktivierungsgruppen

Serviceprogramme mit allgemeingültigen Prozeduren (zum Beispiel Datumsfunktionen) sollten in einer benannten Aktivierungsgruppe ausgeführt werden. Theoretisch kann bei einer benannten Aktivierungsgruppe jeder beliebige Name angegeben werden. Eine einfache Methode, Aktivierungsgruppen und Service-Programme zusammenzufinden ist, die Aktivierungsgruppe und das Service-Programm gleich zu benennen.

Serviceprogramme mit Insert/Update und Delete-Prozeduren in Aktivierungsgruppe *CALLER

Serviceprogramme mit Insert/Update und Delete-Prozeduren sollten mit Aktivierungsgruppe *CALLER erstellt werden, um zusammen mit dem rufenden Programm als Einheit ausgeführt werden zu können.

3.2 RPG

RPG ist DIE Entwicklungssprache, wenn die Vielzahl der Programme und Anwendungen betrachtet wird, die weltweit auf den Systemen AS/400, iSeries und System i betrieben werden. RPG ist die Abkürzung für „Report Program Generator“. Schon aufgrund des Namens lässt es sich erahnen, dass RPG anfänglich für einfache Listprogramme gedacht war, denn zum Zeitpunkt der RPG-Einführung waren Dialoganwendungen noch ein Fremdwort.

Die ursprüngliche Ausrichtung sollte in den Anfängen von RPG dazu dienen, Daten in Reports zu erfassen und diese somit den Anwendern verfügbar zu machen. Heute, mehr als 40 Jahre nach der Ersteinführung, ist aus RPG allerdings eine moderne, umfassende Entwicklungssprache für kaufmännische Anwendungen auf dem System i geworden.

Die Ursprungsvariante von RPG war einzigartig und zeichnete sich durch einige Besonderheiten aus. Eine dieser Besonderheiten ist der strukturelle Aufbau der RPG-Programme, der auch heute noch in den modernen Versionen zum Einsatz kommt. Dabei ist eine bestimmte Reihenfolge der Zeilen im RPG-Code zwingend einzuhalten.

Wie jede andere Programmiersprache verfügt auch RPG über spezifische Eigenarten, die das Erlernen der Sprache nicht gerade vereinfachen. Solche Eigenarten sind neben der bereits erwähnten Struktur des RPG-Programms

- die Verwendung von Bezugszahlen,
- der RPG-Zyklus und
- die Spaltenorientierung.

Hinzu kommt, dass sich die unterschiedlichen Versionen, die auch heute noch sämtlich auf den neuesten System i-Modellen unterstützt werden, zum Teil wesentlich im Aufbau und in der Verwendung der Syntax unterscheiden können. Jungentwickler, die zum Beispiel als Quereinsteiger aus Java, PHP oder vergleichbaren Sprachen den Weg zu RPG finden, haben häufig Einstiegsprobleme. Aber sind die Hürden der RPG-Geheimnisse erst einmal genommen, steht dem System i-Entwickler eine umfassende und moderne Funktionspalette zur Verfügung, mit der er elegant die Anforderungen der Anwender an die System i-IT von heute umsetzen kann.

Die Programmiersprache RPG hat seit ihrer Einführung als einfache Listgeneratorsprache zu Beginn der 60er-Jahre einen langen Entwicklungsweg durchlaufen. Eigentlich sind es unterschiedliche Sprachen, schaut man sich das alte RPG und die neue moderne Free-Format-Version an. Aber IBM hat hier, anders als bei den vielen neuen Namen der Maschine, beharrlich am bewährten Namen RPG festgehalten.

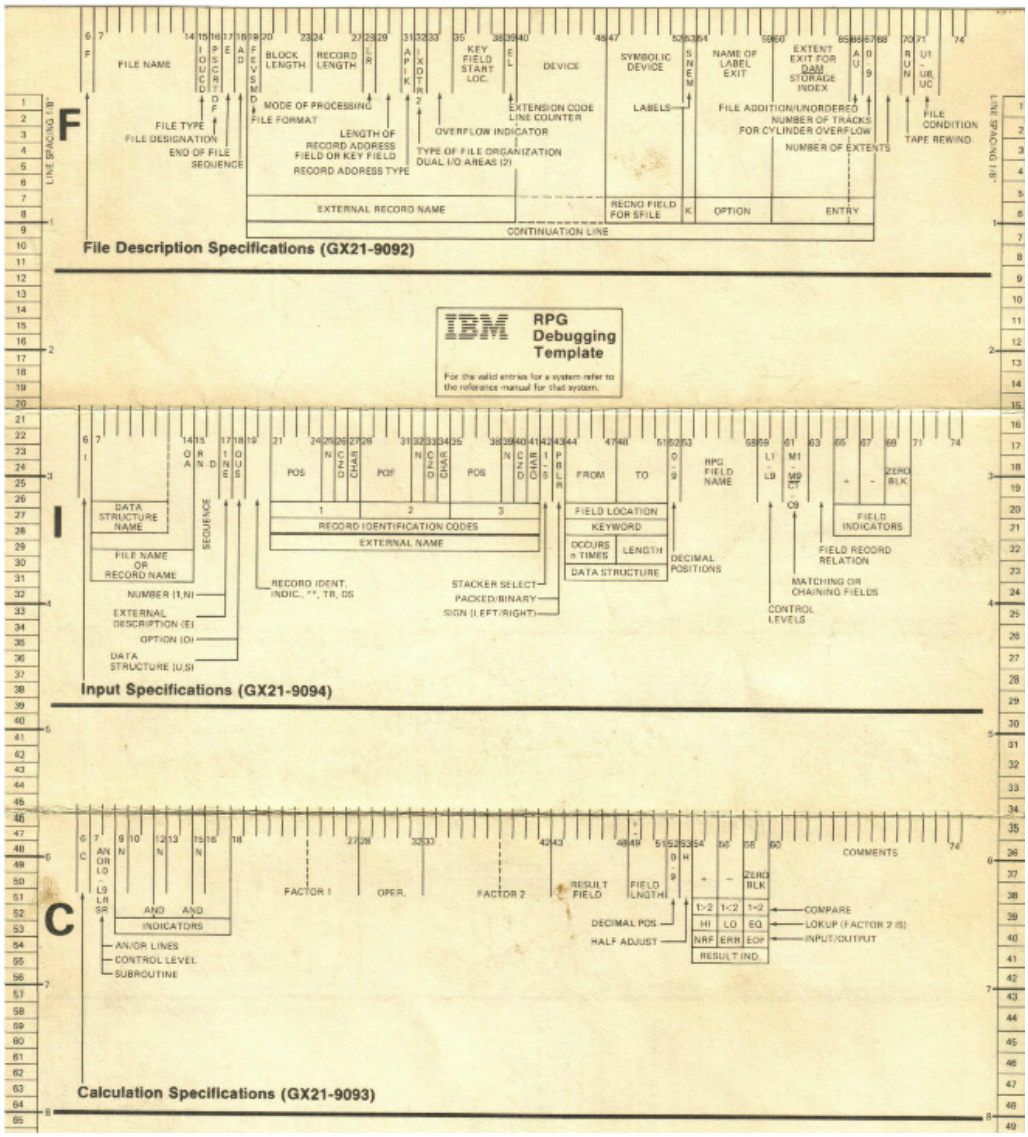
Dass RPG alt sei, ist ebenso ein Gerücht wie das von der „alten“ AS/400. Die Nachteile oder – nennen wir sie besser – die Besonderheiten von RPG sind schlicht Eigenarten. Und Eigenarten hat letztlich jede Programmiersprache. Spaltengebundenheit oder auch die Nutzung von Bezugszahlen im RPG-Zyklus hielten in der Vergangenheit viele Programmierer davon ab, sich mit RPG zu beschäftigen. Dabei sind das lediglich kleine Hürden, die heute kaum noch von Bedeutung sind, denn IBM hat es sich nicht nehmen lassen, RPG über die Jahre mehr und mehr den Anforderungen des Marktes (leider vorzugsweise im Green-Screen-Bereich) anzupassen, sprich neue Techniken zu implementieren und neue Funktionen zur Verfügung zu stellen. ILE-Konstrukte, variable Programmaufrufe in statischer oder dynamischer Form und nicht zuletzt die Free-Format-Variante zeugen von stetiger Weiterentwicklung.

Auch wenn RPG heute immer noch nicht für die Bereitstellung von Webanwendungen ideal ausgelegt ist, so ist dies doch kein Hindernis, das Programm für die Entwicklung performanter und sicherer Anwendungen zu nutzen, die mit modernen Oberflächen ihren Dienst im Intranet oder Internet verrichten. Letztlich ist es eine Frage des Know-hows über Entwicklungssprachen und Entwicklungswerkzeuge im Allgemeinen. Die richtige Kombination aus zum Beispiel RPG-Teilen sowie Java- und XML-Komponenten macht den versierten Entwickler und Softwarearchitekten aus. RPG ist ein Baustein, der sich ideal mit anderen Technologien in Synergie bringen lässt.

Wer sich heute mit RPG beschäftigt, muss sich auch ein wenig mit Historischem auseinandersetzen, also mit den frühen RPG-Versionen. Das ist zumindest dann der Fall, wenn man nicht alle RPG-Anwendungen auf der grünen Wiese startend neu aufsetzen kann.

Nun liegt es an jedem selbst, was er als „früh“ betrachtet. Für den einen Leser mag RPG III noch das täglich Brot sein, während andere längst ILE im Einsatz haben. Ich werde daher versuchen, einen möglichst umfassenden Überblick über die heute noch in Anwendung befindlichen RPG-Versionen zu geben, um Sie in die Lage zu versetzen, bestehende Anwendungen zu lesen, zu warten und im Idealfall durch einen modernen Lösungsansatz im RPG-Umfeld abzulösen.

Spaltenorientierung ist eines der Schlagworte rund um RPG, sie besteht seit den Anfängen des Programms. Sehen Sie sich dazu folgende Abbildung an, die die Spaltenformate einer frühen RPG-Version zeigt.



Codierkarten

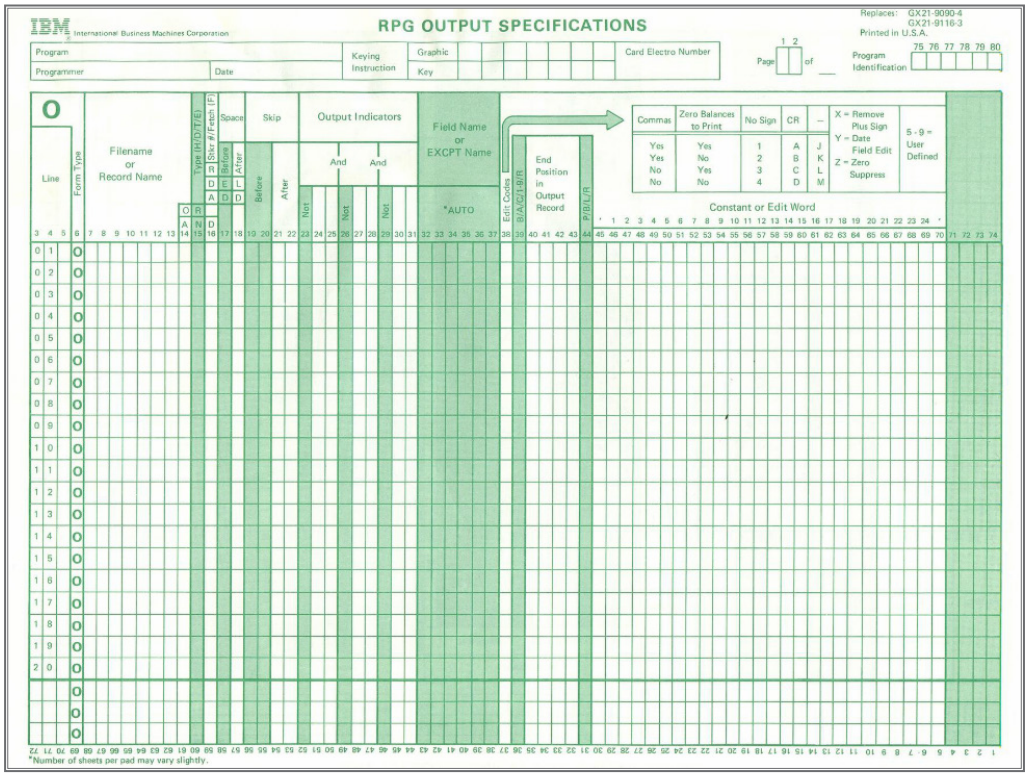
Zugegeben – bei dieser Abbildung handelt es sich um eine sehr frühe Form einer Codierkarte. Sie kommt heute nicht mehr zum Einsatz und soll hier lediglich die Spaltenorientierung auf einen Blick verdeutlichen.

Ursprünglich konzipiert als Sprache zur Ausgabe von Daten in Listenform, wurde RPG im Laufe der Zeit um unterschiedliche Neuerungen erweitert, die mit der Weiterentwicklung der Midrange-Systeme aufkamen. So wandelte sich die Sprache, die dereinst nur Listausgaben erzeugen sollte, allmählich zu DER Universalsprache auf den Midrange-Systemen – auch in Bereichen, die die Datenausgabe auf Bildschirmen oder Batch-Verarbeitungen betreffen.

Auch wenn heute eine Unterstützung für Webanwendungen mit RPG in nativer Form kaum vorhanden ist, so gibt es doch eine Reihe von etablierten Lösungen, die – zusammen mit RPG als einem Hauptbestandteil – Webanwendungen entwickeln und steuern. Dazu später mehr.

Die Spaltenorientierung von RPG ist legendär. Von Kennern geschätzt, von Laien belächelt, bietet sie natürlich sowohl Vorteile als auch Nachteile. Im weiteren Verlauf des Buchs folgt zu den Besonderheiten des Fixformats bzw. des spaltenorientierten Formats mehr. Die nachfolgende Abbildung veranschaulicht zunächst einmal recht schön die Spalten und deren Verwendung.

In frühen RPG-Jahren war dies die Planungs- und Codiervorlage der Anwendungsentwickler.



RPG-Ausgabespezifikationen

Die beiden vorhergehenden Abbildungen zeigen die „Urzeiten“ von RPG. Da wir das Jahr 201x und nicht mehr die 1980er-Jahre schreiben, sollten Sie sich nicht davon abschrecken lassen.

RPG ist heute mehr als 40 Jahre alt, was aber nicht heißt, dass es sich um eine „alte“ Programmiersprache handelt. Denn schaut man sich die heutige Version von RPG an, dann ist RPG zu einer wirklich modernen und auch in der heutigen Zeit leistungsfähigen Sprache gewachsen.

RPG kann, richtig verstanden und auch richtig eingesetzt, eine sehr moderne Anwendungsentwicklungssprache sein, die sich auf dem System i eignet. Die Besonderheit an RPG-Programmen ist, dass in existierenden Umgebungen häufig die verschiedenen RPG-Varianten, beginnend mit dem klassischen RPG III über RPG IV bis hin zum ILE- und Free-Format-RPG, gefunden werden. Diese verschiedenen Varianten werden an späterer Stelle vorgestellt.

Auch im Fall von RPG zeigt es sich, dass ein bestimmter Vorteil der AS/400 und der Nachfolgemodelle gleichzeitig ein massiver Nachteil sein kann: Bietet man den Entwicklern die Wahlmöglichkeit zwischen Steinzeitwerkzeugen und modernsten Technologien, dann werden viele die alten Technologien wie bisher weiternutzen wollen, weil sie ihnen bereits bekannt sind. Wozu auch Neues lernen? Solange diese Option besteht, wird RPG sich nur schwer den Namen auf dem Markt machen, den RPG verdient – nämlich den einer modernen Anwendungsentwicklungssprache.

Welche Möglichkeiten bietet uns RPG heute?

Vergleicht man die erste Version von RPG mit der neuesten, modernen Ausführung des Free-Format RPG IV, die auf dem System i verfügbar sind, dann lässt sich feststellen: Die Entwicklung, die sich in all den Jahren beständig fortgesetzt hat, ist wirklich enorm!

Vielleicht hätte IBM gut daran getan, den Namen von RPG in seiner jüngsten Form der Entwicklung anzupassen und zu ändern – aber dies wäre dann vermutlich ähnlich verlaufen wie bei den vielen „Neubenennungen" des Systems. Also bleibt es in unserem Sprachgebrauch schlicht bei RPG.

Die Kunst ist es nun, die verschiedenen Versionen von RPG zu differenzieren und für sich selbst abzuschätzen, welche RPG-Versionen und -Funktionen für den individuellen Bedarf sinnvoll sind.

Eine moderne RPG-Anwendung kommt in der Regel um den Einsatz von ILE nicht herum. ILE ist eine ganz neue Denkweise in der RPG-Welt, die im späteren Verlauf genauer behandelt wird. –

Anmerkung!

ILE ist eine Philosophie oder eine Technologie, die in die Lage versetzt, auf dem System i Programm- oder Anwendungskomponenten zu kombinieren und durch Wiederverwendbarkeit und modulares Bausteine Anwendungen zu formieren. Dabei lassen sich Programmkomponenten der verschiedensten Arten (CL, Cobol, Java, RPG etc.) zu ausführbaren Programmeinheiten zusammenfügen.

Diese Technologie ist nicht mit dem erweiterten Faktor 2 zu verwechseln, der zeitnah mit ILE angekündigt wurde. Beim erweiterten Faktor 2 hat IBM den Typ der Programmcodes von RPG auf RPGLE geändert. Zwar lässt die Erweiterung durch LE einen direkten Zusammenhang mit ILE erahnen, jedoch ist das reine Erweitern einer Programmcodeschablone bei weitem nicht das, was ILE darstellt. Dazu später mehr.

Es folgt die Erklärung einiger Begrifflichkeiten rund um RPG, die immer wieder genannt werden und für das Grundverständnis von RPG unerlässlich sind.

Der erweiterte Faktor 2

Mit der Einführung des sogenannten erweiterten Faktors 2 wurde eine wesentliche Abgrenzung gegenüber den früheren Release-Versionen von RPG erzielt. Mit Hilfe dieser Erweiterung, die unter anderem für die Angabe von spezifischen und erweiterten Rechenoperationen eingesetzt werden kann, können RPG-Programmierer jetzt unter Verwendung von EVAL- und If-Anweisungen erweiterte Ausdrücke nutzen. Mit diesen Neuerungen sind verschiedene Grenzen gefallen, die den RPG-Entwicklern in früheren Zeiten immer wieder Mehraufwand bescherten.

Ein wesentlicher Schritt in Richtung Modernisierung der Programmiersprache ist es, dass verschiedene mathematische Operanten (+, –, *, /) in den Ausdrücken verwendet werden. Weitere als Ausdruckerweiterung zu verwendende Operanten sind:

- gleich (=)
- kleiner als (<)
- größer als (>)
- kleiner gleich (<=)
- größer gleich (>=)
- ungleich (<>)

Das mathematische Exponentensymbol (**) wurde als neuer mathematischer Operant in RPG eingeführt. Zwar wird die Exponentialrechnung in kaufmännischen Anwendungen nicht sehr oft genutzt, so erleichtert die Neuerung doch die Bewältigung von Neuanforderungen enorm.

Eine weitere Neuerung in der Version RPG IV sind die Klammerkonstrukte. Die Unterstützung durch runde Klammern bietet den Anwendungsentwicklern die Möglichkeit der Verschachtelung mathematischer Anweisungen. Diese Anweisungen wurden bereits damals in anderen Programmiersprachen unterstützt – es war also eine Notwendigkeit, dass IBM diese auch in der neuen Version von RPG verfügbar machte.

Eines der zuvor genannten Symbole, das +-Zeichen, hat in RPG IV eine weitere Bedeutung: Es wird als Verkettungssymbol verwendet. Mit diesem Symbol können RPG-Programmierer zum Beispiel Daten vom Typ Character zusammenfügen und als String ausgeben. Zwar hat es solche Möglichkeiten auch vorher gegeben, nur waren diese wesentlich umständlicher.

Built-In-Functions
Für System i-Anwendungsentwickler waren die Built-In-Functions eigentlich nicht wirklich neu, denn vergleichbare Funktionen gibt es auch in der Steuersprache des OS/400 – der CL-Sprache. So sind z. B. die häufig verwendeten Funktionen %SST(Substring) und %Bin(Binary) in CL verwendete Built-In-Functions.

Die eigentliche Idee zu den in RPG verwendeten Built-In-Functions kommt aus dem Bereich der C-Programmierung. Eine Built-In-Function stellt dabei entweder eine Anweisung mit der wahlweisen Verwendung von Parametern dar, die normalerweise Daten eines vorher festgelegten Datentyps an der Stelle des Programms zurückgibt, an der die Built-In-Function im Programm integriert wurde. So prüft zum Beispiel die RPG IV %Eof Built-In-Function das Erreichen des Dateiendes nach einer Leseanweisung. Dadurch entfallen die sonst erforderliche Angabe und die Abfrage eines Dateiende-Anzeigers, wie er häufig in Form einer Bezugszahl verwendet wird. Ähnlich kann die %Found Built-In-Function nach einer CHAIN-Anweisung verwendet werden, um abzufragen, ob ein Lesezugriff erfolgreich war.

Die Anzahl der in RPG unterstützten Built-In-Functions wurde seit der Einführung bis zur heutigen Version auf 76 Built-In-Functions erweitert, wobei einige davon eine alternative Möglichkeit zur Verwendung älterer Funktions-Codes darstellen. Beispiele hierfür sind Scan (Built-In-Function %Scan) und Subst (Built-In-Function %Subst). In einigen Fällen bieten die moderneren Built-In-Functions mehr Flexibilität und mehr Funktionsumfang als die älteren Operations-Codes, für die sie eingesetzt werden können. So können zum Beispiel mit Hilfe der Built-In-Function %Subst sowohl ein Sub-String als auch ein Quell-String eines Ergebnisses verarbeitet werden, wogegen der ursprüngliche Befehl SUBST (Sub-String) lediglich den Quell-String verarbeiten konnte.

Subprozeduren
IBM hat noch eine weitere leistungsfähige Neuerung in RPG IV integriert. Mit der Unterstützung von Subprozeduren haben Anwendungsentwickler die Möglichkeit, komplexe Programme in kleinere, einfacher zu wartende Teilbereiche aufzuspalten und damit eine Art „eigenerstellte“ Built-In-Functions zu realisieren. IBM hat in der ersten Version der Subprozeduren das Konzept der lokalen Variablen realisiert, die innerhalb der Subprozeduren verwendet werden können. Ebenfalls neu war die Option, rekursive Aufrufe in Verbindung mit Subprozeduren verwenden zu können.

Weitere Veränderungen

Mit dem Erscheinen von RPG IV hat IBM zudem einige Altlasten des RPG/400 (Vorgänger des RPG IV) eliminiert. So wurden zum Beispiel die Längenbeschränkungen von acht Stellen für Dateinamen und Satzformatnamen auf zehn Stellen erweitert. Auch die maximal mögliche Länge für Feldnamen wurde erhöht – und zwar wesentlich: von zehn auf 4.096 Stellen! Die maximale Größe von Character-Feldern wuchs von 256 auf 32.767 Bytes. Ähnliche Veränderungen wurden bei gepackten, numerischen Feldern realisiert, deren maximale mögliche Feldlänge von 30 auf 63 angehoben wurde. Welchen Sinn haben solch lange Feldnamen? Die Antwort zeigt sich, wenn man sich mit qualifizierten Feldnamen beschäftigt, deren Verwendung einige Vorteile mit sich bringt.

Doch damit nicht genug – neben den zuvor genannten Datentypen gab es einige weitere, neue, die erstmals in RPG IV realisiert wurden: der Pointer-Datentyp zur Aufnahme der Adresse der Daten bzw. der Prozeduren, der Objekt-Datentyp für die Verarbeitung von Referenzierungen zu einem Java-Objekt sowie verschiedene Datums-, Zeit- und Timestamp-Datentypen.

Allein diese neuen Datentypen und die zum Teil wesentlichen Erweiterungen der bisherigen Begrenzungen in bestehende Datentypen zeigen, welch signifikanten Änderungen bereits an der Basis von RPG vorgenommen wurden. IBM hat diese Änderungen meiner Meinung nach durchgeführt, da für die iSeries eine modernere Anwendungsentwicklung geboten werden muss, mit Hilfe derer die Anwender moderne und leistungsfähige Anwendungen im Bereich der Midrange-Server realisieren können.

Es ist sicher nicht ganz falsch, die „alten“ RPG-Versionen als eine Art „Dinosaurier“ der Programmiersprachen zu bezeichnen. Die Unterscheidungen zwischen den alten, in die Jahre gekommenen RPG-Versionen und den „modernen“ Anwendungssprachen wie C, C++ und Java waren zum Teil enorm. Diese Differenzierungen und auch Einschränkungen, die die alten RPG-Versionen mit sich brachten, sind mit dem Einsatz der modernen und neuen RPG-Versionen quasi nicht mehr vorhanden. Mit RPG IV ist es IBM gelungen, an die Leistungsfähigkeiten moderner Sprachen heranzukommen, wenn nicht gar diese heimlich in der Leistungsfähigkeit zu überholen. Damit stehen den Anwendungsentwicklern alle Tore für die Realisierung von modernen Anwendungen weit offen.

Sinnvolle „Nebeneffekte“

Wie programmiere ich richtig? Diese nicht leicht zu beantwortende Frage stellen sich sicher viele Anwendungsentwickler.

Selbst wenn Sie 1969 ein RPG-Programm geschrieben haben, in dem die Programmlogik über RPG-Zyklen und 99 Bezugszahlen abgebildet war, wird IBM Ihnen heute nicht sagen, dass Sie etwas falsch machen – selbst dann nicht, wenn diese Programme noch immer im Einsatz sind. IBM bietet nach wie vor die Möglichkeit der Verwendung und des Einsatzes aller bisher existierenden RPG-Versionen.

Die Hardware hat sich in der Vergangenheit durchschnittlich alle sechs bis acht Jahre deutlich verändert. IBM hat stets die Möglichkeit geboten, ältere Programmversionen mit neuen Maschinen zu unterstützen. Damit verbunden ist allerdings auch die Tatsache, dass die moderne Programmiersprache RPG IV heute in vielen Fällen Altlasten mit sich herumschleppt, deren Ursprünge zum Teil 40 Jahre zurückliegen – oder verwenden Sie wirklich keine RPG-Zyklen oder Bezugszahlen mehr in Ihren Programmen? Die Realität ist, dass auch mit dem modernen RPG IV noch heute die Programmiertechniken der 1960er-Jahre existieren.

Als IBM die Version IV von RPG angekündigt hatte, bestand die Hoffnung, dass es endlich einen Schnitt zwischen Vergangenheit und moderner Welt der Anwendungsentwicklung geben würde. Unternehmen, deren Anwendungen nicht erweitert oder modernisiert werden müssen, können mit den bisherigen Programmen und Methoden weiterarbeiten. Und solche Unternehmen, die neue Anforderungen haben, können die modernen Techniken verwenden. Ich selbst hatte gehofft, dass diese neue RPG-Version endlich den Teufelskreis durchbrechen würde.

Vielleicht wären wir lediglich in der Lage, neue Instruktionen, wie zum Beispiel EVAL, zu verwenden, statt mit ADD, SUB, MULT und DIV zu arbeiten, also Funktionen, die eigentlich in die Geschichtsbücher gehören. Es wäre schön, die Rechendefinitionen zu eliminieren und sie mit den neuen Definitionsspezifikationen zu ersetzen. Die Praxis sieht jedoch anders aus. Es gibt ein ständiges Nebeneinanderherlaufen der alten Techniken parallel mit den modernen Methoden. Schade, denn RPG IV ist eine wirklich moderne Programmiersprache, die eine Vielzahl an Funktionen und Techniken bietet, welche für die Entwicklung von modernen Anwendungen benötigt werden und diese neuen Techniken werden meist parallel zu alten Funktionen eingesetzt.

Neue Programme brauchen nicht mit den Altlasten früherer Release-Versionen erstellt zu werden. Wozu noch Primary-Dateien oder Variablen verwenden, die in Kalkulationen definiert wurden? Das ist sicher alles richtig. Dennoch sind im modernen RPG IV Bestandteile der alten RPG-Zyklen enthalten, und diese werden auch benötigt. Wie sonst könnten LR- und RT-Bezugszahlen ihre Arbeit ausführen?

3.2
Seite 10

3.2.1 RPG-Versionen

Bevor in die Details eingestiegen wird, folgt hier eine kurze Übersicht der RPG-Versionen.

RPG II
RPG II wurde mit dem System /3 bereitgestellt und auch von den Modellen /32, /34 und /36 unterstützt.

RPG III
RPG III wurde zusammen mit dem System /38 ausgeliefert und war die erste offizielle Standard-RPG-Variante für das System AS/400 und wird auch heute noch von den aktuellen System i-Modellen unterstützt.

RPG 400
Bei RPG 400 handelt es sich um die RPG III-Version, die auf dem System AS/400 ausgeliefert wurde – sie ist folglich mit dem RPG III gleichzusetzen.

Diese RPG-Versionen werden alle mit der Quellenart RPG definiert. Die Compiler bieten auf dem System i die für die Quellenart RPG erforderlichen Hilfen und Prüfungen.

Alle zuvor genannten RPG-Versionen sind Klassiker – Altversionen, die jedoch heute noch auf den modernen System i-Modellen unterstützt werden. IBM hat die Weiterentwicklung dieser Versionen allerdings vor geraumer Zeit eingestellt. Alleine deshalb ist es dringend zu empfehlen, Neuentwicklungen grundsätzlich nur mit den modernen und zeitgemäßen Versionen des RPG durchzuführen.

RPG IV
Mit der Release-Version V3R1M0 des Betriebssystems kündigte IBM auch die Weiterentwicklung von RPG in Form von RPG IV an. Zusammen mit dieser neuen Version wurde die Quellenart RPGLE eingeführt. RPG IV stellt die konsequente Weiterentwicklung von RPG III dar und beinhaltet unter anderem neue Anweisungen, den erweiterten Faktor 2 und die Unterstützung von ersten Built-In-Functions (BIFs).

RPG IV – offiziell mit ILE RPG bezeichnet – erschien erstmals Ende 1994 mit der Einführung des OS/400 Release V3R1. IBM hatte bereits vorher das „Integrated Language Environment“ (ILE) eingeführt – aber viele hatten auf die Integration von ILE in RPG gewartet.

Ergänzung 31/2014

3.2.1
Seite 2

Dabei handelt es sich um drei wesentliche Neuerungen, die mit der Version V3R1M0 in RPG erschienen sind:

- den erweiterten Faktor 2,
- die „Built-In-Functions“ (kurz BIF) und
- die Subprozeduren.

Was hier in einem Nebensatz erwähnt wurde, ist die Grundlage für ILE. Jedoch ist ILE nicht gleichzusetzen mit RPG IV. Zwar ist RPG IV die Voraussetzung für ILE, aber ILE ist eine komplexe Technologie, ein Programmkonzept oder sogar eine Entwicklungsphilosophie, die weit mehr darstellt als eine Form der Codierung.

Später ein wenig mehr zu ILE-RPG.

Schaut man sich die Historie von RPG an, dann lässt sich diese gut visualisieren (siehe folgende Abbildung).

RPG Überblick

Beginnend mit dem Ur-RPG hat IBM den Weg der RPG-Versionen konstant weiter entwickelt, sodass heute mit Free-Format-RPG, ILE und RPG OpenAccess die jüngsten Vertreter der Programmiersprache genutzt werden können.

Aber was ist ein Programm? Was ist daran interessant? Wohl weniger die Anwendung selbst als der Weg dorthin. Letztlich ist natürlich das RPG-Programm, das eine Tätigkeit ausführt, von Bedeutung – und nicht der Programmcode oder irgendwelche Werkzeuge, die Entwickler einsetzen können, um die Wünsche ihrer Anwender umzusetzen. RPG-Programme bestehen aus:

- dem Quellcode (Text – mit speziellen Editoren zu verwalten),
- dem ausführbaren Programmobjekt (Ergebnis einer erfolgreichen Umwandlung Kompilierung).

Hier folgt eine kurze Auflistung der RPG-„Besonderheiten“:

- am häufigsten verwendete Programmiersprache auf AS/400, iSeries und System i,
- in älteren Varianten spaltenorientiert,
- in neueren Varianten Free-Format,
- Zyklusverwendung,
- Bezugszahlen und
- LR (Indikator zur Kennzeichnung des Programmendes).

3.2.1 Seite 4

3.2.2 RPG-Formate

Die Spaltenorientierung von RPG ist eine der Besonderheiten von RPG. Anwender machen jedoch keinen Unterschied darin, ob das Programm in einem spaltenorientierten, älteren Codeformat geschrieben wurde oder ob der Anwendungsentwickler die moderne Free-Format-Variante fürs Codieren genutzt hat.

Wichtig für den Entwickler aber ist, dass die beiden Varianten auch heute noch verfügbar sind und dass die Wartung der älteren Programme Kenntnisse im Fixformat RPG unbedingt voraussetzt. Gerade RPG-Neulinge stehen beim Codieren oft vor unerklärlichen Problemen, die sich letztlich durch eine Stellenverschiebung im Quelltext darstellen lassen. So exotisch diese Spaltenorientierung auch ist, so sehr hat sich bei Tausenden von Anwendungsentwicklern über viele Jahre das Schreiben eines stabilen und funktionalen Codes in genau diesem Format bewährt.

Die Herausforderung von heute lautet: Anwendungsentwickler benötigen RPG-Kenntnisse sowohl im Fix- als auch im Free-Format.

3.2.2
Seite 2

3.2.2.1 Fixformat RPG

Eine Eigenart, die auch heute noch im direkten Zusammenhang mit RPG steht, ist die Spaltenorientierung, das heißt, dass der Quellcode exakt in den vorgegebenen Spalten einzugeben ist. Dabei haben die einzelnen Spalten in Abhängigkeit der Bestimmungen unterschiedliche Bedeutungen. Dazu später mehr.

Ältere RPG-Programme wurden im Fixformat geschrieben. Erst nach Jahrzehnten hat IBM diese Tatsache als Nachteil erkannt und die Möglichkeit geschaffen, den RPG-Code auch im freien Spaltenformat eingeben zu können.

Nachfolgende Abbildung zeigt exemplarisch einen Ausschnitt eines Fixformat-RPG-Programms.

```
     *****************************************************************
     *  1. RPG Beispiel im Fix-Format
     * Copyright (c) J. Zeig / ZeBIS 2014
     *****************************************************************
    H DftActGrp(*NO) ActGrp(*CALLER)
    H DatFmt(*EUR) TimFmt(*HMS)
    H option(*srcstmt:*NoDebugIO)
     * Rechenbestimmungen
    d Text            S             50A   inz('1. Anzeige im RPG')

     ** ---------------------------------------------------------
     * Rechenbestimmungen
     ** ---------------------------------------------------------

    c* Anzeige der Nachricht
    c     text          dsply

    c* Ende des Programms
    c                   eval      *inlr = '1'
```

RPG Codebeispiel im Fixformat

3.2.2.2 Free-Format RPG (erste Version)

Die zuvor genannten RPG-Versionen zeichnen sich alle durch eine spaltenorientierte Codierung aus. Das bedeutet, dass der Anwendungsentwickler sich strikt und konsequent an die vorgegebenen Spalten halten muss, in denen die Programmcodes einzugeben sind. Diese Spaltenorientierung ist es, die RPG den Beigeschmack einer alten, starren und unflexiblen Programmiersprache verliehen hat.

IBM hat einen ersten Schritt in Richtung RPG-Moderne gemacht, indem eine teilweise Free-Format-Struktur bereitgestellt wurde. „Teilweise" bedeutet in diesem Zusammenhang, dass die Spalten nicht zu 100 Prozent verschwunden sind, sondern dass die freie Eingabe lediglich auf bestimmte Bestimmungen beschränkt wurde. „Free-Format" ist die Bezeichnung für die Codierform, welche ab der Version V5R1M0 des Betriebssystems eingeführt wurde.

Das Free-Format erlaubt die flexible und spaltenfreie Eingabe von Programmcodes, wobei der RPG-Fluss und die Anordnung der RPG-Komponenten in Form der Anweisungsebenen erhalten bleiben. Unabhängig von der spaltenorientierten oder freien Form ist die Quellart RPGLE. Das freie Format steht nicht für die alten RPG-Quellen zur Verfügung, sondern ist auf den Einsatz mit RPGLE- oder SQLRPGLE-Quellen ausgelegt.

Nachfolgende Abbildung zeigt einen kleinen Ausschnitt aus dem Free-Format RPG-Quellcode.

```
// Ausgabe Bildschirm: Eingabe Zahlen
/FREE
 dow F3_Ende = '0';
  exfmt fmt00;

  // Definition der Befehlstaste F3 (*in03)
  // und Schleifenbeginn IF
  if not F3_Ende;

   // Berechnen Variable - Addition
  if F5_Add;
   F00Ergeb = F00Zahl1 + F00Zahl2;
  endif;
```

Beispiel Free-Format

3.2.2
Seite 4

Das Codebeispiel soll an dieser Stelle lediglich einen ersten Eindruck vermitteln – Details zum Free-Format werden im späteren Verlauf des Buches gezeigt.

Programmierer können bei allen RPGLE-Versionen das freie Format alternativ oder parallel zum spaltenorientierten Fixformat einsetzen.

Bei der Codierung ist zu beachten, dass die Codeblöcke des freien Formats mit der Anweisung „/Free“ beginnen und mit der Anweisung „/End-Free“ enden.

IBM machte in Sachen Free-Format einen großen Schritt, doch dieser war letztlich nicht konsequent genug. Das Wegfallen einiger antiquarischer Anweisungen aus der spaltenorientierten Zeit ist ein Tribut, das gezollt werden musste. Die Beschränkung der Freiheit auf die Rechenbestimmungen war letztlich eine Unverständlichkeit, denn gerade die jüngste Form des Free-Formats, die mit der Version 7 veröffentlich wurde, zeigt das, was man vor Jahren schon hätte erwarten dürfen oder müssen.

Die Unmengen an Fixformatcodes gepaart mit der Notwendigkeit, bestimmte Anweisungen, die in der neuen Free-Format -Version nicht mehr unterstützt werden, zumindest temporär weiternutzen zu müssen, führt zu einer Mischform der Codeeingabe: Fixformat im Wechsel mit Free-Format.

Nachfolgende Abbildung zeigt dazu ein Beispiel.

```
        // Ende der do-Schleife
       enddo;

      /end-free
     C                   GOTO       Adresse
      /free

       // Programmende
       *inlr = *on;

      /END-FREE
```

Mit der jüngsten RPG-Version sind einige Anweisungen zurückgezogen worden und stehen den Programmieren nun nicht mehr zur Verfügung. Ein Beispiel dafür ist das Fehler der Anweisung MOVE, welche durch moderne Alternativen ersetzt wurde. Auch die berüchtigte GOTO-Anweisung, die im vorangehenden Codebeispiel exemplarisch enthalten ist, wird in der Free-Format-Version nicht mehr unterstützt.

Mit dem Wegfallen solcher Anweisungen geht IBM endlich den konsequenten Weg in Richtung Moderne … wenngleich mit vielen Pausen …

Mit der Einführung des Free-Formats ist RPG der Sprung in die Klasse der modernen Entwicklungssprachen gelungen, mit der wichtige, neue Funktionen zur Verfügung stehen. Diese aktuellen Erweiterungen in RPG bieten die Möglichkeit, Programme zu erstellen, die einfacher zu warten, leichter zu lesen und besser zu verstehen sind – ein Vorteil, der nicht nur jungen Entwicklern zugutekommt, sondern auch den RPG-Kennern mit der Zeit viele Vorteile bringt.

So groß der Schritt vom langjährigen Fixformat hin zum Free-Format auch war, es war lediglich ein halbherziger Schritt. Denn genau genommen hätte das Format „freies Rechenbestimmungsformat“ heißen müssen, da lediglich die Rechenbestimmungen in der ersten Free-Format-Variante von der Spaltenbindung befreit waren. Einige Jahre später waren auch die übrigen Anweisungsbereiche „frei“. Dazu später mehr.

3.2.2.3 Free-Format RPG (finale Version)

Das Free-Format RPG hat eigentlich erst seit Anfang 2013 seinen Namen auch wirklich verdient. Mit den Neuerungen rund um die Version IBM i 7.1 kündigte IBM die derzeit aktuellste RPG-Variante an – das „echte" Free-Format RPG. Mit dieser Version sind nun auch die Anweisungen neben den Rechenbestimmungen im freien Format codierbar. Damit entfallen Spaltenbindungen nahezu komplett – eine Erleichterung für Neueinsteiger und universelle Entwicklungen.

Verwenden Sie wenn irgend möglich die neueste Version des Free-Formats RPG! Auch wenn die Anwender nicht merken, in welcher Variante das ausführbare Programm codiert wurde, so erleichtert es durch den größeren Funktionsumfang und durch die flexible Codiermöglichkeit die Lesbarkeit für Sie, für Kollegen und auch für zukünftige Programmierer.

Der beherzte Einsatz moderner Technologien des RPG ist ein Muss, wenn mit RPG-Anwendungen die Anforderungen des Jahres 201x gekonnt umgesetzt werden sollen.

Die Stärken von RPG liegen klar im gemeinsamen Einsatz mit der Datenbank DB2 der AS/400 und der Nachfolgemodelle (heute Power System i). RPG dient im kaufmännischen Umfeld der Datenlistenbearbeitung, das heißt, die Hauptaufgabe besteht im Lesen, Berechnen und formatierten Ausgeben von Datenbankdateien.

Die Fix-/Free-Formate sind letztlich nur für den Anwendungsentwickler von Bedeutung – der Endanwender merkt in der Regel nicht, welche Form der Programmierung verwendet wurde.

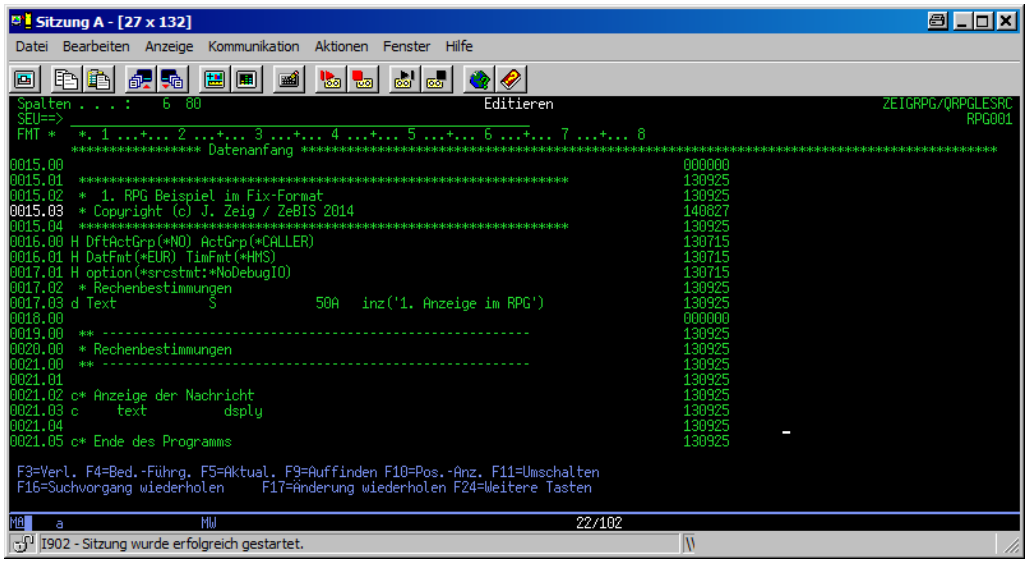
Im Anschluss an die Darstellung der Historie von RPG folgt eine gezielte Beschäftigung mit den RPG-Varianten, da diese häufig in Kundenumgebungen zum Einsatz kommen. Leider finden sich dort auch im großen Maße die RPG III-Derivate, gefolgt von RPGLE-Varianten und dem RPG-Free-Format. Da dieses Buch einen möglichst umfassenden Einblick in die Welt der Anwendungsentwicklung des System i bieten möchte, müssen die gängigen RPG-Varianten so behandelt werden, dass der Leser ein Verständnis über die nutzbaren RPG-Derivate gewinnt.

3.2.3 RPG-Editoren

RPG ist eine Programmiersprache wie viele andere auch. Jede Programmiersprache muss in irgendeiner Form codiert werden, was mit einem Editor ausgeführt wird. Der RPG-Editor ist ein gewöhnlicher Texteditor zur Eingabe der Befehle und Codeanweisungen und das Handwerkzeug des Anwendungsentwicklers.

IBM stellt fürs Codieren unterschiedliche Editoren zur Verfügung, welche die verschiedenen RPG-Varianten unterstützen. Die wohl verbreitetste Editorversion stellt SEU dar.

Folgende Abbildung zeigt eine geöffnete RPG-Quelle im Fixformat im Green Screen Editor SEU.



Codebeispiel RPG-Fixformat im SEU-Editor

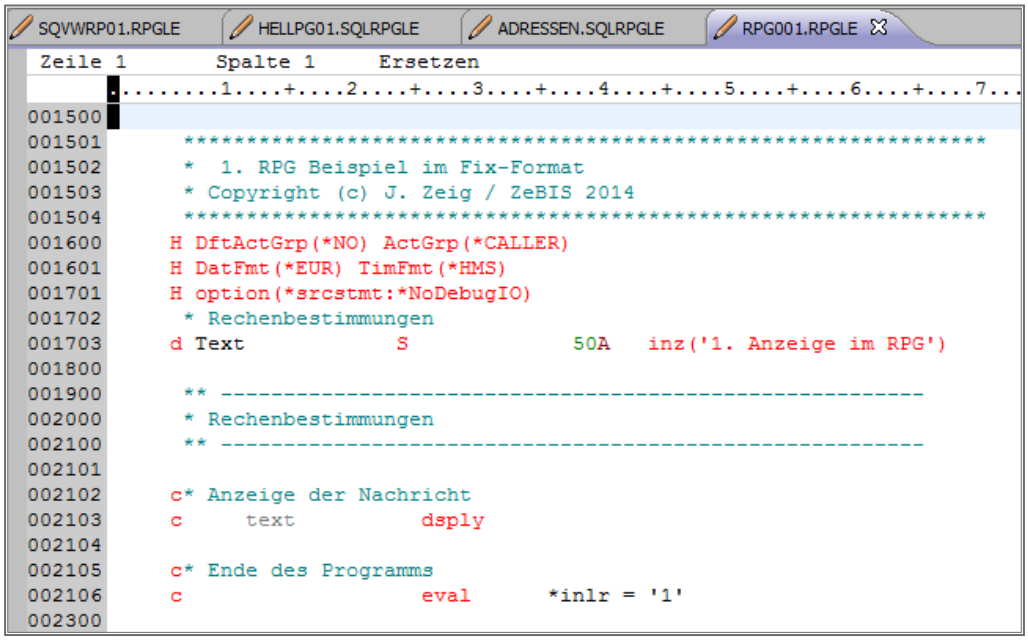
Der SEU-Editor unterstützt alle auf dem System i zur Verfügung stehenden RPG-Versionen vom RPG III über den RPG IV bis hin zum Free-Format RPG. Allerdings ist dessen Einsatz auf die 5250-Emulation beschränkt.

Neuerungen an Editorfunktionen finden sich jedoch seit einigen Jahren nicht mehr im SEU, dessen Entwicklung zugunsten des Nachfolgers der 5250-basierten Entwicklungsumgebung eingestellt wurde. Die Editorfunktionen aus dem Rational Developer für System i sind die würdigen Nachfolger der lange Jahre dominierenden SEU-Funktionen. Neuentwicklungen, insbesondere beim Einsatz von Free-Format RPG sowie der Erstellung und Verwaltung von ILE-basierten Bausteinen, sind mit dem LPEX-Editor der Rational-Familie leicht umzusetzen.

3.2.3
Seite 2

An dieser Stelle sollen nicht die Editoren im Detail behandelt werden, sondern es soll lediglich ein Einblick in die Möglichkeiten des Codierens von RPG gegeben werden. Details über die Editoren finden Sie an anderer Stelle in diesem Buch.

Folgende Abbildung zeigt dasselbe Codebeispiel, das zuvor auch im SEU-Editor gezeigt wurde.



Codebeispiel RPG-Fixformat im LPEX-Editor

Im zuvor gezeigten Beispiel findet sich eine moderne Versionen des Editors, mit dem der RPG-Quelltext eingesehen und auch bearbeitet werden kann. Dabei handelt es sich um den sogenannten LPEX-Editor aus der Rational-Familie, der auf der Eclipse-Plattform basiert.

Dieser vollständig in Java geschriebene Editor steht für Windows-Systeme zur Verfügung und erlaubt das Codieren von System i-Quellen. Farbliche Differenzierungen der einzelnen Codebestandteile gehören natürlich ebenso zur Grundausstattung wie Bedienerhilfen, Einrückoptionen etc.

Nach der Einleitung rund ums Thema RPG folgt die Beschäftigung mit dem Codieren von RPG.

3.2.4 RPG-Bestimmungen

RPG basiert auf verschiedenen Regeln, die von den Programmierern strikt einzuhalten sind, damit der Programmcode letztlich vom Compiler verarbeitet und ein Objekt auf dem System i erstellt werden kann.

Eine der Regeln bezieht sich auf die sogenannten Bestimmungen, die den zeilenorientierten Ablauf des Programmcodes festlegen. Dabei ist zwingend die von IBM vorgegebene Reihenfolge einzuhalten. Im folgenden Abschnitt soll ein Einblick in die verschiedenen RPG-Bestimmungen gegeben werden mit Bezug auf das Fixformat.

Für RPG-Neulinge sind an dieser Stelle die Beschreibungen der verschiedenen Bestimmungen wichtig. Die Bedeutung der einzelnen Spalten und der damit verbundenen Möglichkeiten ist wichtig für das allgemeine Verständnis der spaltenorientierten Programmierung mit RPG. An dieser Stelle erhalten Sie einen ersten Einblick in die Bedeutung der Spalten für die unterschiedlichen Bestimmungen.

H-Bestimmungen (Header)

- Allgemeiner Beschreibungsbereich für das Programm
- Optionale Angabe

Hier ein Codebeispiel für H-Bestimmungen:

```
H DatFmt(*EUR) TimFmt(*HMS)
H option(*srcstmt:*NoDebugIO)
H dftactgrp(*no) actgrp(*caller)
```

Der Aufbau der H-Zeilen unterliegt den folgenden Spaltenvorgaben:
Spalte 6: H (Header)
Spalte 7–80: Angabe der Schlüsselwörter

Im RPG-Editor zeigen sich die Spalten in der Übersicht wie folgt:

```
.....HKeywords++++++++++++++++++++++++++++++++
```

3.2.4
Seite 2

F-Bestimmungen (Files)

- Angaben der im Programm zu verwendenden Dateien
  - Name der Datei
- Verwendungsart (Eingabe, Ausgabe, Update, Kombination)
- Format (extern oder intern beschrieben)
- Einheit (Platte, Drucker, Arbeitsstation)

Der Aufbau der C-Zeilen unterliegt den folgenden Spaltenvorgaben:

Spalte 6:	C (Calculation)
Spalte 7 – 16:	Dateiname
Spalte 17:	Dateityp
Spalte 18:	Dateiverwendung
Spalte 19:	Dateiende
Spalte 20:	Hinzufügen neuer Sätze
Spalte 21:	Reihenfolge
Spalte 22:	Dateiformat
Spalte 23 – 27:	Satzlänge
Spalte 28:	Verarbeitung in Grenzen
Spalte 29 – 33:	Länge des Schlüsselfeldes
Spalte 34:	Satzadressart
Spalte 35:	Dateiorganisation
Spalte 36 – 42:	Einheit
Spalte 43:	Leer
Spalte 44 – 80:	Schlüsselwörter

Im Editor ist folgende Spaltenlegende dargestellt:

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++
```

Nachfolgende Abbildung zeigt Beispiele für Angaben in den Dateibestimmungen.

```
FDomtraErr uf a E           K Disk
FDomtraErL2uf   E           K Disk    Rename(RDomtraerr:Rdomtral2)
FDomprot   if a E           K Disk
```

D-Bestimmungen (Definitionen)

- Definitionsbereich für Variable, wie zum Beispiel
  - Felder,
  - benannte Konstanten,
  - Datenstrukturen,
  - Feldgruppen,
  - benannte Indikatoren.

Der Aufbau der D-Zeilen unterliegt den folgenden Spaltenvorgaben:

Spalte 6: D (Definition)

Spalte 7–21: Variablenname

Spalte 22: Datenstrukturdefinition
- *Leer* Normale Definition
- E Extern beschriebene Definition

Spalte 23: Art der Datenstruktur
- *Leer* Keine Datenstruktur
- S Programmstatusdatenstruktur
- U Datenbereichsdatenstruktur

Spalte 24–25: Deklarationsart
- *Leer* Unterfeld einer Datenstruktur, Parameter im Prototypfeld in PI
- C Konstante
- DS Datenstruktur
- PI Procedure Interface
- PR Prototyp für Prozeduraufruf
- S Eigenständiges Feld oder Tabelle

Spalte 26–32: Positionswert VON
- *Leer* Längendefinition ist im BIS-Feld angegeben
- nnnn Startposition Datenstrukturunterfeld

Spalte 33–39: Positionswert BIS bzw. Länge
- *Leer* An anderer Stelle definiert
- nnnn Länge, wenn VON-Wert fehlt

3.2.4
Seite 4

Spalte 40: Interne Datenart (Auszug)

Leer	An anderer Stelle definiert
A	Alphanumerisches Feld
B	Binäres Feld
D	Datumsfeld
F	Gleitkommafeld
I	Integer, ganze Zahl mit Vorzeichen
N	Indikator (*on/*Off)
P	Gepacktes Dezimalfeld
S	Gezontes Dezimalfeld
T	Zeitfeld
U	Unsigned, ganze Zahl ohne Vorzeichen
*	Pointer

Spalte 41 – 42: Anzahl Dezimalstellen
Spalte 44 – 80: Schlüsselwörter

Die Spalteneinteilung zeigt sich im RPG-Editor wie folgt:

```
.....DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++
```

Nachfolgend finden sich einige Beispiele für die Verwendung von D-Bestimmungen.

```
DF01NMESperre             56      56N
DF01NMECursor             57      57N

 * ----------------------------------------------------------------
 * Variablen deklarieren
 * ----------------------------------------------------------------
D g_retCode       S               1S 0
D g_bMatNrOK      S                N
D g_KzVerArb      S               1a   inz

 // ---------------------------------------------------------
 // Skalare Daten
 // ---------------------------------------------------------
Drsn              S               3S 0 inz

d ArtI_K        e ds                   extName(ARTIKELL1:*key)
d                                      qualified
d ArtI_I        e ds                   extName(ARTIKELL1:*Input)
d                                      qualified
d ArtI_O        e ds                   extName(ARTIKELL1:*Output)
d                                      qualified
```

I-Bestimmungen (Eingabedefinitionen) (nach Möglichkeit nicht mehr verwenden!)

- Definitionsbereich für Eingabedefinitionen
  - NUR IN AUSNAHMEFÄLLEN VERWENDEN – VERALTET!!!

C-Bestimmungen (Rechenbestimmungen)

- DER eigentliche RPG-Programmbereich
- Gibt es nur im Fixformat (im Free-Format sind Rechenbestimmungen der Standard)
- Prozesssteuerung
- Bedingungsprüfung
- Datenmanipulation
- Benutzerinteraktion
- Berechnungen
- …

Bei den Rechenbestimmungen muss der Aufbau hinsichtlich des alten RPG III und der weiterentwickelten RPG-Fixformatversion mit dem erweiterten Faktor 2 unterschieden werden.

Nachfolgend soll das Augenmerk bewusst auf die Version mit dem erweiterten Faktor 2 gelegt werden.

Im Editor finden sich folgende Spaltenlegenden für RPG IV:

```
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
```

Der Aufbau der C-Zeilen unterliegt den folgenden Spaltenvorgaben:

Spalte 6:	C (Calculation)
Spalte 7–8:	Gruppenstufe
Spalte 9–11:	Bezugszahlen
Spalte 12–25:	Faktor 1
Spalte 26–35:	Anweisung der Erweiterung
Spalte 36–49:	Faktor 2
Spalte 50–63:	Ergebnisfeld
Spalte 64–68:	Feldlänge
Spalte 69–70:	Dezimalstellen
Spalte 71–76:	Ergebnisbezugszahlen

```
c* Anzeige der Nachricht
c     text          dsply

c* Ende des Programms
c                   eval      *inlr = '1'
```

3.2.4
Seite 6

O-Bestimmungen (Ausgabedefinitionen) (nach Möglichkeit nicht mehr verwenden!)

- Definitionsbereich für Ausgabedefinitionen, wie zum Beispiel programmbeschriebene Druckerdateien

```
Odrucker   e             kopfex01
o
```

P-Bestimmungen (Prozedurdefinitionen)

Definitionsbereich für eine Prozedur innerhalb eines RPG-Programms. Die Prozedurangaben bestehen aus der Angabe des Beginns (B) und der Angabe des Endes (E).

Prozeduren können im RPG-Code zur Gliederung eingesetzt werden und spielen unter anderem in der ILE-Welt eine besondere Rolle.

```
     h NoMain

      * Funktion-Anfang
     p DemoProz1       B                   Export

      * Funktion Interface mit Rückgabewert!!
     D DemoProz1       PI                  3a
     d Firma                               3A
     d Teil                               10A

      * Definitionen allgemein
     d var             S                   1A

      * beliebiges Einbinden weiterer Prozeduren:
     c                   callp(e)   DemoProz2(Firma)

      * Funktionsende mit Angabe "return" und Rückgabewert
     c                   return     Firma

      * Funktion-Ende
     p DemoProz1       E
```

Zu beachten!

Die Reihenfolge der Bestimmungen ist zu beachten!!

So müssen F-Bestimmungen zwingend vor D- und C-Bestimmungen angegeben werden!

3.2.4.1 Kommentare

Auch wenn eine Priorisierung der Kommentarzeilen an dieser Stelle befremdlich wirkt, ist deren Bedeutung doch sehr groß. Denn jeder gute Programmcode sollte über ein Mindestmaß an Kommentaren und Erläuterungen verfügen. Mit dem sinnvollen Einsatz von Kommentaren wird die Lesbarkeit von Programmcodes erleichtert, und er trägt dazu bei, sich nach einer Pause in den Code und die einzelnen Teilbereiche schneller und leichter einarbeiten zu können. Dies erhöht die Effektivität.

Wenn es auch keine Bestimmungen für Kommentarzeilen gibt, dürfen diese in keinem Programmcode fehlen, denn es handelt sich hierbei einfach um eine Sache des guten Programmierstils.

Die Kommentarzeilen in RPG können an jeder beliebigen Stelle im Programmcode eingebunden werden. Es gibt hier keine Rangfolge und auch keine Reihenfolge.

Kommentarzeilen sind abhängig von den beiden Codierformaten (Fix-Format und Free-Format) unterschiedlich anzugeben.

Es folgen nun zwei Beispiele, bei denen zwischen dem Fix-Format RPG und dem Free-Format RPG differenziert wird.

Fix-Format-Kommentare
Das erste Beispiel zeigt die Verwendung von Kommentarzeilen im Fix-Format-RPG-Programmcode. Die Regeln dazu lauten:

- Die Angabe einer Kommentarzeile kann überall im RPG-Programmcode erfolgen.
- Keine Berücksichtigung von Reihenfolgen.
- Kommentarzeilen werden vom Kompiler NICHT übersetzt und NICHT interpretiert.
- Kommentarzeilen dienen der besseren Lesbarkeit des Programms.
- Die Kommentarzeile muss im Fixformat-Code an der Spalte 7 ein Asteriskzeichen (*) haben.
  Alle auf das Asteriskzeichen folgenden Zeichen werden als Kommentare interpretiert.
- Im Programmcode können mehrere aufeinander folgende Zeilen als Kommentare erfasst werden.

3.2.4
Seite 8

```
Zeile 14      Spalte 8     Einfügen  13 Änderungen
      ..... *..1....+....2....+....3....+....4....+....5....+....6....+....7....+.
001501      C*****************************************************************
001502      C*
001503      C*  RPG GANZ KLASSISCH
001505      C*
001506      C* Copyright (c) J. Zeig / ZeBIS 2013
001507      C*****************************************************************
001508      C*
001509      F* Ein Beispiel für einen Kommentar auf der Datei-Bestimmungsebene
001510      D* Ein Beispiel auf D Ebene
001511       * ein Beispiel ohne Angabe einer Bestimmungsangabe - aber dennoch ein
001512       * gültiger Kommentar
001513       * ach sind die Moves schoen!
001514       *
001515       *
001516      C                   MOVE 'ALT'     VAR     10
002300      C                   MOVE '1'       *INLR
```

Kommentarbeispiel im Fix-Format RPG

Wie auf oben stehender Abbildung ersichtlich, beginnen die Kommentarzeilen mit einem Asteriskzeichen in der Spalte 7. Manchen Kommentarzeilen sind die Bestimmungsarten der jeweiligen Zeile vorangestellt, z. B. Zeilen Nr. 1501 bis 1510.

```
C*****************************************************************
C*
F* Ein Beispiel für einen Kommentar auf der Datei-Bestimmungsebene
D* Ein Beispiel auf D Ebene
```

Kommentarbeispiel mit Bestimmungsart

Andere Kommentarzeilen lassen die Art der Zeilenbestimmung in der Spalte 6 vermissen – das Startzeichen für die Kommentarzeile steht dabei alleine in der Spalte 7 gefolgt vom Kommentar.

```
 * ein Beispiel ohne Angabe einer Bestimmungsangabe - aber dennoch ein
 * gültiger Kommentar
```

Kommentarbeispiel ohne Bestimmungsart

Es ist gleich, welche der beiden Eingabevarianten für Kommentare im Fix-Format bevorzugt werden, denn im Endeffekt sind beide in der Wirkung identisch. Es kommt also nicht darauf an, ob in der Spalte 6 die Art der Zeilenbestimmung mitgegeben oder ob lediglich die Pflichtstartangabe für den Kommentar in der Spalte 7 mit dem *-Zeichen eingegeben wird.

Wie zuvor erwähnt, sind alle nach dem Kommentarstartzeichen folgenden Einträge in der Zeile von der Interpretation her Kommentarzeilen. Das bedeutet auch, dass die Zeile 801 im folgenden Beispiel zwar auf den ersten Blick wie eine Rechenbestimmungszeile aussieht, dass sie aber aufgrund der Angabe des *-Zeichens in der Spalte 7 einen Kommentar darstellt.

```
000700
000800       * Ein Beispiel für einen Kommentar
000801     c*                    eval      Verarb = '*'
000900     C                     eval      Var = 'eval'
000901     C                     eval      *inlr = '1'
000902
```

Kommentarvarianten Fix-Format

Der in den Rational-Produkten enthaltene Editor zeigt die Kommentare in einer abweichenden Farbdarstellung getrennt von den eigentlichen „echten" Codezeilen an. Der SEU-Editor allerdings lässt eine optische Differenzierung missen, wie auf nachfolgender Abbildung zu sehen.

```
0008.00  * Ein Beispiel für einen Kommentar
0008.01 c*                    eval      Verarb = '*'
0009.00 C                     eval      Var = 'eval'
0009.01 C                     eval      *inlr = '1'
0009.02
```

Kommentar im SEU

Nur bei genauem Hinsehen fällt die vermeintliche Rechenbestimmung als Kommentarzeile auf.

Ein weiterer Grund für die Nutzung der Rational-Funktionen!

Tipp:

Wenn Sie eine Codezeile auskommentieren und damit inaktivieren, dann verwenden Sie zur besseren Lesbarkeit mehrere aufeinander folgende Asteriskzeichen. Nachfolgende Abbildung zeigt dazu ein Beispiel.

3.2.4
Seite 10

```
C*****************************************************
C*
C*  RPG fast ein bisschen moderner
C*
C* Copyright (c) Joerg Zeig / ZeBIS 2013
C*****************************************************

 * die nachfolgende Zeile ist auskommentiert
C*++++              MOVE      'ALT'          VAR
 * und noch eine Form eines Kommentars:
 *****              MOVE      '1'            *INLR
```

Übersichtliche Kommentarkennzeichnung

In der Abbildung finden Sie unterschiedliche Formen von Fix-Format-Kommentaren. Ausschlaggebend ist jeweils das Asteriskzeichen an der Stelle 7. Alleine diese Position markiert eine Zeile als Kommentarzeile im Fix-Format-RPG-Programmcode.

Free-Format-Kommentare
Die Kommentare im Free-Format unterscheiden sich wesentlich von denen im Fix-Format.

Jeder Kommentar muss mit zwei aufeinander folgenden Schrägstrichen (//) beginnen. Alle darauf in der Zeile folgenden Zeichen werden dann als Kommentare interpretiert. Die erste Eingabestelle ist die Position 8.

Folgende Abbildung zeigt Beispiele für die Verwendung von Kommentaren im Free-Format RPG.

```
008400      //-------------------------------------------------------------------------
008500      //    V A R I A B L E N
008600      //-------------------------------------------------------------------------
008700
009600
009700      /eject
009800      //*************************************************************************
009900      //          -----------------
010000      //         M A I N - L I N E
010100      //          -----------------
010200      //*************************************************************************
```

Kommentar im Free-Format

Die beiden führenden //-Zeichen definieren den Kommentar. Welche Zeichen danach folgen, ist für die Kommentarzeile unerheblich.

Wichtig sind hier die einleitenden doppelten Schrägstriche. Im Free-Format können diese an jeder beliebigen Stelle in der Programmzeile (ab Spalte 7) angegeben werden.

```
000500     C* Copyright (c) Joerg Zeig / ZeBIS 2013
000600     C*****************************************************************
000601     D var             S              1A
000700
000800      * Ein Beispiel für einen Kommentar
000801     c*                   eval      Verarb = '*'
000900     C                    eval      Var = 'eval'
000901     C                    eval      *inlr = '1'
000902
000903      /FREE
000904      // Beispiele für Kommentare im Free-Format
000905        var = *blank;   // Feld initialisieren ein Text in einer Zeile
000906
000907      // und noch ein text//  var = 1;
000908
000909        *inlr = *on;
```

Kommentarversionen in LPEX Editor

Vorhergehende Abbildung zeigt verschiedene Beispiele für Kommentarzeilen – sowohl im Fix-Format RPG als auch in der freien Variante. Auf Zeile Nr. 905 finden Sie ein Beispiel für eine aktive Programmzeile mit nachfolgendem Kommentar.

Auch das ist ein Pluspunkt für die Verwendung der Rational-Werkzeuge: Die aktiven Zeilenbereiche werden schwarz dargestellt und lassen den aktiven Programmcode bereits optisch erkennen. Der im rechten Bereich folgende Kommentar „Feld initialisieren …:“ wird grün dargestellt.

3.2.4.2 Steuerbestimmungen

Im vorhergehenden Kapitel wurde eine erste Übersicht zu den unterschiedlichen Bestimmungsformen im RPG-Programm gegeben.

Da für RPG-Anwendungsentwickler die Reihenfolge der Bestimmungen von wesentlicher Bedeutung ist, werden im Folgenden die unterschiedlichen Formen der Bestimmungen erläutert sowie deren Einsatzgebiete und Besonderheiten aufgezeigt.

Doch vorerst zu den Steuer- oder Header-Bestimmungen (kurz: H-Bestimmungen).

Wie schon der Name andeutet, handelt es sich hierbei um Steuerangaben fürs RPG-Programm. Header-Anweisungen stehen in allen RPG-Versionen (Fix-Format und Free-Format) zur Verfügung.

Vielen Anwendungsentwicklern ist die Bedeutung der Steuerbestimmungen gar nicht bewusst, und es werden nur wenige Angaben getätigt. Spätestens mit der Verwendung von ILE jedoch tritt deren Bedeutung in den Vordergrund. Beinahe unbemerkt hat IBM im Zusammenhang mit den letzten Release-Versionen neue Steuerbestimmungen bereitgestellt, mithilfe derer sich das allgemeine Verhalten des Programms beeinflussen lässt.

Folgende Abbildung zeigt exemplarisch ein Beispiel für den Einsatz von Steuerbestimmungen.

```
H/TITLE G E N S R V 0 0 1 - Serviceprogramm ZeBIS
H DATFMT(*ISO) DECEDIT('0,') OPTION(*NODEBUGIO)
H BNDDIR('QC2LE')
H NOMAIN
H COPYRIGHT('ZeBIS, 2009-2015, Kreuztal')
```

Beispiel für Steuerbestimmungen

Steuerbestimmungen sind für viele RPG-Entwickler der alten Schule eher lästig, weshalb sie oft stiefmütterlich behandelt oder gar ganz weggelassen werden. Im Grunde ist das auch in Ordnung, kommt doch ein RPG-Programm auch ohne Steuerbestimmungen aus. Allerdings bringt sich der RPG-Entwickler doch um eine Reihe von Vorteilen:

- Datumsbehandlung,
- Zeitdarstellung,
- Copyright-Angaben,
- Debug-Einstellungen,
- Editierverhalten
- und vieles mehr.

Globale Einstellungen lassen sich auf Basis der Steuerbestimmungen definieren, womit z. B. auch das einheitliche Verhalten des Programms in Bezug auf die Datumsaufbereitung, die Darstellung von numerischen Werten oder die Verwendung von Währungssymbolen verbunden ist.

Welches Verhalten ein RPG-Programm zeigen soll, kann auf unterschiedliche Weise hinterlegt werden. Entweder lassen sich die Angaben mittels spezieller Datenstrukturen (RPGLEHSPEC) oder DFTLEHSPEC festlegen oder individuell auf Source-Datei-Ebene in den Steuerbestimmungen der Quellteildatei. Zu Letzterem mehr.

Wie bereits erwähnt, ist die Reihenfolge der Steuerbestimmungen innerhalb eines RPG-Programms wichtig und zwingend vorgeschrieben.

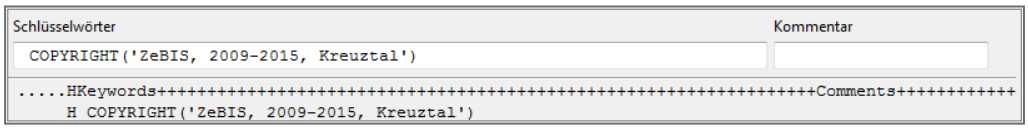
Deshalb müssen Steuerbestimmungen, wenn sie denn in einem Programm angegeben werden, an erster Stelle im Programmcode stehen. Ausgenommen hiervon sind Kommentarzeilen.

Die Angaben der Steuerbestimmungen können im Fix-Format RPG an jeder beliebigen Spalte (ab Spalte 8) codiert werden.

Ergänzung 33/2015

3.2.4
Seite 14

Zunächst ist der Aufbau der Steuerbestimmungen im Fix-Format RPG gezeigt.



Aufbau der Steuerbestimmungen

In der Abbildung ist ersichtlich, dass die Steuerbestimmungen aus

- dem H in Spalte 6,
- den Schlüsselwörtern (der Haupteingabebereich für die Steuerbestimmungen)in den Spalten 7 bis 80 und
- dem Kommentar (optional)

bestehen.

Da die Vielzahl der Schlüsselwörter hier nicht umfassend behandelt werden kann, wird zumindest auf die gängigen eingegangen.

Grundsätzlich können die Schlüsselwörter in den Steuerbestimmungen ohne oder zwangsweise bzw. wahlweise mit Parametern angegeben werden. Wenn ein oder mehrere Parameter für eine Steuerbestimmung anzugeben sind, dann müssen diese in Klammern eingeschlossen werden.

Detailinformationen zu den jeweiligen Schlüsselwörtern und deren Parametern sind auf den IBM-Webseiten oder in der Bedienerhilfe der Rational-Werkzeuge für System i zu finden. Hier ist eine Übersicht zu den möglichen Schlüsselwörtern aus der Bedienerhilfe gegeben:

Keywords

- ALLOC(*STGMDL | *TERASPACE | *SNGLVL)
- ACTGRP(*STGMDL | *NEW | *CALLER | 'activation-group-name')
- ALTSEQ{(*NONE | *SRC | *EXT)}
- ALWNULL(*NO | *INPUTONLY | *USRCTL)
- AUT(*LIBRCRTAUT | *ALL | *CHANGE | *USE | *EXCLUDE | 'authorization-list-name')
- BNDDIR('binding-directory-name' {:'binding-directory-name'…})
- COPYNEST(number)
- CCSID(*GRAPH : parameter | *UCS2 : number | *CHAR : *JOBRUN)
- COPYRIGHT('copyright string')
- CURSYM('sym')
- CVTOPT(*{NO}DATETIME *{NO}GRAPHIC *{NO}VARCHAR *{NO}VARGRAPHIC)
- DATEDIT(fmt{separator})
- DATFMT(fmt{separator})
- DEBUG{(*INPUT | *DUMP | *XMLSAX | *NO | *YES)}
- DECEDIT(*JOBRUN | 'value')
- DFTACTGRP(*YES | *NO)
- DFTNAME(rpg_name)
- ENBPFRCOL(*PEP | *ENTRYEXIT | *FULL)
- EXPROPTS(*MAXDIGITS | *RESDECPOS)
- EXTBININT{(*NO | *YES)}
- FIXNBR(*{NO}ZONED *{NO}INPUTPACKED)
- FLTDIV{(*NO | *YES)}
- FORMSALIGN{(*NO | *YES)}
- FTRANS{(*NONE | *SRC)}
- GENLVL(number)
- INDENT(*NONE | 'character-value')
- INTPREC(10 | 20)
- LANGID(*JOBRUN | *JOB | 'language-identifier'')
- MAIN(main_procedure_name)
- NOMAIN

3.2.4
Seite 16

- OPENOPT (*NOINZOFL | *INZOFL)
- OPTIMIZE(*NONE | *BASIC | *FULL)
- OPTION(*{NO}XREF *{NO}GEN *{NO}SECLVL *{NO}SHOWCPY *{NO}EXPDDS *{NO}EXT *{NO}SHOWSKP) *{NO}SRCSTMT) *{NO}DEBUGIO) *{NO}UNREF
- PRFDTA(*NOCOL | *COL)
- SRTSEQ(*HEX | *JOB | *JOBRUN | *LANGIDUNQ | *LANGIDSHR | ‚sort-table-name‘)
- STGMDL(*INHERIT | *SNGLVL | *TERASPACE)
- TEXT(*SRCMBRTXT | *BLANK | 'description')
- THREAD(*CONCURRENT | *SERIALIZE)
- TIMFMT(fmt{separator})
- TRUNCNBR(*YES | *NO)
- USRPRF(*USER | *OWNER)

Während einige Steuerbestimmungen schon zu Zeiten des RPG III und der OPM-Modelle (klassische Form der RPG-Anwendungsentwicklung) zur Verfügung gestanden haben, haben sich andere erst im Zuge der ILE-Bereitstellung in die Auflistung eingereiht.

Gerade in der ILE-Entwicklung sind Steuerbestimmungen wichtig. Unter anderem lässt sich damit die Angabe eines Binderverzeichnisses (BNDDIR) oder die Angabe steuern, dass eine Programmkomponente nicht über eine MAIN-Prozedur verfügen muss (NOMAIN).

Andere Steuerangaben regeln z. B. das Verhalten der Darstellung von Datumswerten, Zeitwerten oder Währungen.

Beispiele dafür sind:

DATFMT(fmt{separator})
Bei der Darstellung von Datumswerten kann der Anwendungsentwickler in den Steuerbestimmungen allgemeine Vorgaben tätigen. Mit dem Schlüsselwort DATFMT können individuell die Darstellung und die Behandlung von Datumsfeldern geregelt werden.

Wird das Schlüsselwort DATFMT nicht verwendet, dann nutzt das RPG-Programm das ISO-Datumsformat.

Nachfolgend ein Beispiel für das Codieren von DATFMT im Euro-Format:

```
h datfmt(*EUR)
h datedit(*DMY.)
```

Beispiel für DATFMT

DATEDIT(fmt{separator})
Mit dem Schlüsselwort DATEDIT wird definiert, wie numerische Felder beim Einsatz des Editierschlüssels Y darzustellen sind. Der Editierschlüssel Y stellt einen numerischen Wert im Datumsformat dar. Gültige Werte sind:

*DMY,

*MDY,

*YMD.

Per Standard wird als Trennzeichen der Schrägstrich (/) verwendet. Soll ein anderes Trennzeichen genutzt werden, dann muss dieses separat angegeben werden.

```
h datedit(*DMY.)
```

Beispiel für DATEDIT

3.2.4
Seite 18

OPTION(*{NO}XREF *{NO}GEN *{NO}SECLVL *{NO}SHOWCPY *{NO}EXPDDS *{NO}EXT *{NO}SHOWSKP) *{NO}SRCSTMT) *{NO}DEBUGIO) *{NO}UNREF

Der Schlüsselwert OPTION hat große Bedeutung hinsichtlich der Effektivität. Es gibt Phänomene wie ein im Debug ausgeführtes Programm, das bei jedem Datei-Handling erst mehrere Male auf der Datei steht, bis der Debug auswertbar bzw. nutzbar ist. Auch werden das Öffnen und Schließen von Dateien bei jeder Ausführung des Debuggers mitberücksichtigt, wenn auf dem System keine anderen allgemeinen Vorgaben getätigt wurden. Gut ist das nur so lange, wie man als Anwendungsentwickler auch Optionen hat, mit der Datei und dem aktuellen Verarbeitungsstatus derselben arbeiten zu können, was nicht immer der Fall ist.

Folgende Einstellung bewirkt, dass das Öffnen und Schließen von Dateien nicht mehr als Debug-Aktionen ausgeführt werden, womit Debug-Funktionen effektiver werden.

```
h option(*SRCSTMT:*NoDebugIO)
```

Beispiel für Option

Weitere Einsatzgebiete für das Schlüsselwort OPTION sind:

- *EXPDDS
  Bewirkt eine ausführliche Darstellung der Feldinformationen extern beschriebener Dateien in der Feldauflistung. Zudem werden Schlüsselinformationen mit ausgegeben. Insbesondere bei der Verwendung des 5250-basierten SEU-Editors ist das ein nützliches Tool. Die Verwendung dieses Schlüsselwortes ist beim Einsatz von LPEX-Editoren weniger von Bedeutung – diese liefern vergleichbare Informationen auf Basis der Gliederungssicht.
- *SHOWCPY
  Wenn in einer Quelle mit eingebundenen/Copy-Elementen gearbeitet wird, dann lässt sich über diese Option steuern, ob die Copy-Strecken in der Wandlungsliste eingebunden werden sollen, was die Lesbarkeit erleichtert.

Insbesondere bei der Behandlung von SQL-Daten sind Nullwerte wichtig. Wie diese in einem RPG-Programm oder Modul behandelt werden sollen, wird mit der Angabe des Schlüsselwortes ALWNULL geregelt.

ALWNULL(*NO | *INPUTONLY | *USRCTL)
Felder mit Nullwerten und deren Behandlung lassen sich auf der Ebene der Quellenteildatei mittels der Anweisung ALWNULL steuern.

Wenn der Wert *NO gesetzt ist, dann können innerhalb des Programms keine Felder aus extern beschriebenen Dateien verarbeitet werden, die Nullwerte aufweisen. Beim Verarbeiten solcher Daten werden keine Inhalte zurückgegeben, und das Programm läuft in eine Fehlerroutine.

Mit der Angabe *INPUTONLY können Sätze mit Nullwertfeldern eingelesen werden. Die Programmausführung wird in diesem Fall wie üblich fortgesetzt.

Mit der Angabe *USRCTL wird dem Kompiler mitgeteilt, dass man sich explizit um die Behandlung der Nullwertfelder kümmert.

SRTSEQ(*HEX | *JOB | *JOBRUN | *LANGIDUNQ | *LANGIDSHR | 'sort-table-name')
Mit der Angabe von SRTSEQ kann die Sortierung von Daten beeinflusst werden, was z. B. im Einsatz von internationalen Anwendungen Sinn macht. Denn dort sind einige Sortierungen abweichend vom deutschen Standard.

COPYRIGHT('copyright string')
Manchmal ist es erforderlich, dass Copyright-Informationen in einer Quelle hinterlegt werden. Damit diese nicht bloß als reiner Text im Programmcode, sondern auch im zu erstellenden Objekt enthalten sind, wird das Schlüsselwort COPYRIGHT verwendet.

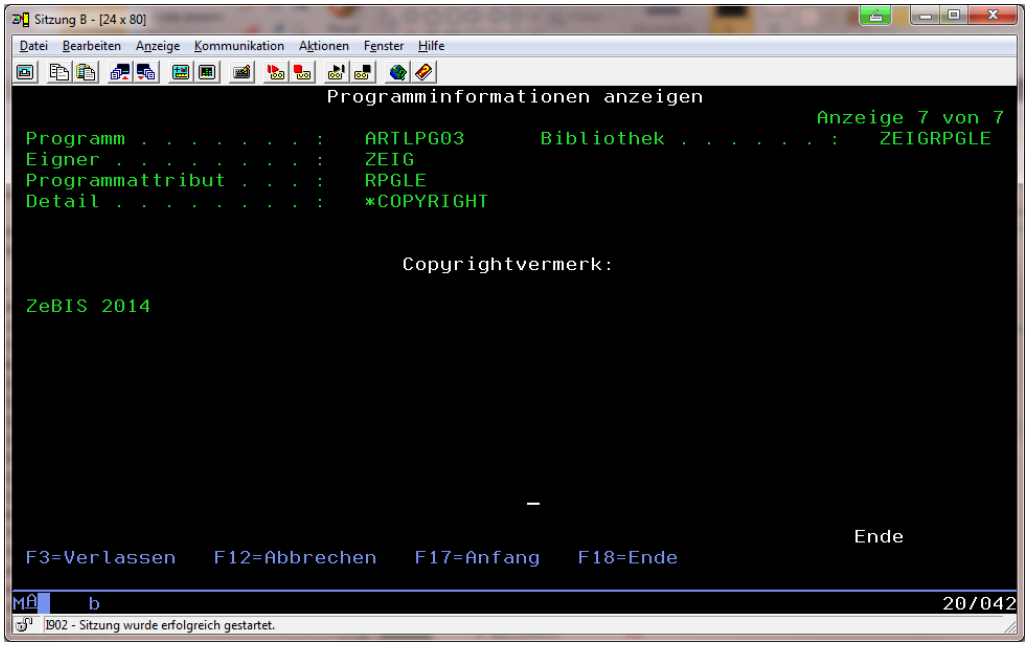
```
H COPYRIGHT('ZeBIS 2014')
H DATEDIT(*DMY.)
```

Beispiel für die Angabe von COPYRIGHT

Dieser Eintrag bewirkt, dass im erstellten Objekt der Copyright-Vermerk unentfernbar eingebunden wird.

3.2.4
Seite 20

Folgende Abbildung zeigt ein Beispiel für den im Programmobjekt eingebundenen Copyright-Vermerk.



Copyright-Hinweis im Programmobjekt

Nachdem Sie einen Einblick in die möglichen Schlüsselwörter für das allgemeine Steuern eines RPG-Programms erhalten haben, werden Ihnen einige Schlüsselwörter aufgezeigt, deren Einsatz im Hinblick auf ILE von Bedeutung ist. Das Verhalten von Programmen oder Modulen, das Zuweisen von Aktivierungsgruppen oder auch die Bereitstellung von RPG-Funktionen im WebService über PCML sind Beispiele für den Einsatz der Steuerbestimmungen in RPG. Doch Performance-Steigerungen oder Ansätze moderner Anwendungen lassen sich seit einiger Zeit auch mit RPG und Threads abbilden. Die Thread-Behandlung ist ein weiteres Beispiel für die Steuerbestimmungen.

Im Folgenden sind einige dieser ILE-bezogenen Steueranweisungen aufgezeigt.

ACTGRP(*STGMDL | *NEW | *CALLER | 'activation-group-name')
Die Ausführung eines Programms unterliegt den Bestimmungen des Betriebssystems und der dazugehörigen Komponenten. Aktivierungsgruppen sind ein wichtiges Element davon. Die Zuordnung zu einer Aktivierungsgruppe kann über das Schlüsselwort ACTGRP in den Steuerbestimmungen erfolgen.

Folgende Abbildung zeigt exemplarisch die Verwendung des Schlüsselworts ACTGRP.

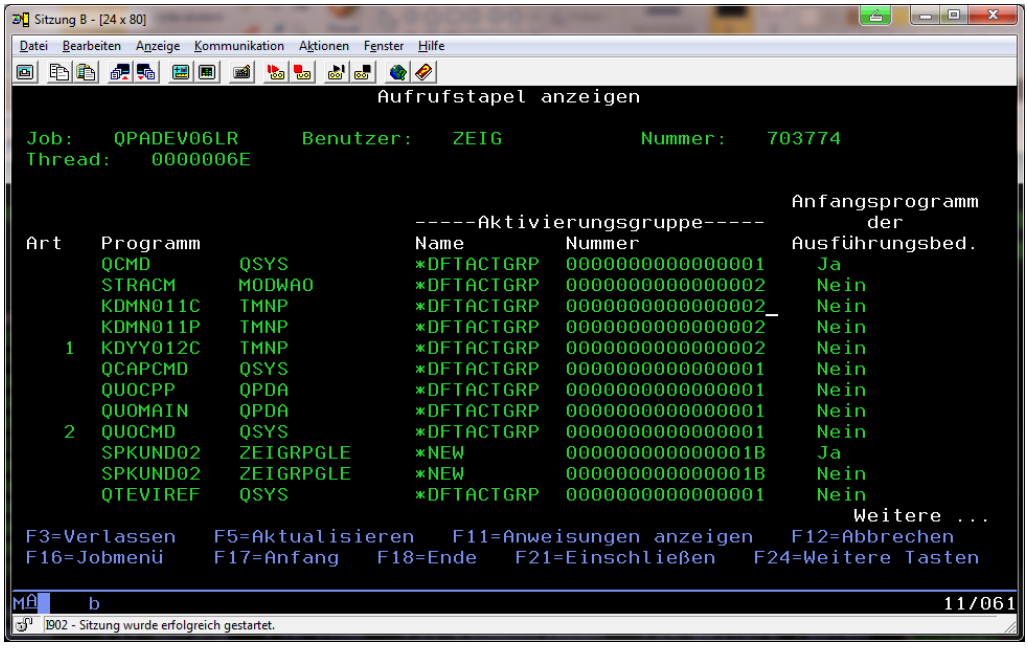
```
h option(*SRCSTMT:*NoDebugIO)
h actgrp('VERTRIEB')
```

Beispiel für ACTGRP

Wenn ein Programm gezielt in einer Aktivierungsgruppe ausgeführt werden soll, dann gibt es zwei Optionen dafür:

- Angabe einer benannten Aktivierungsgruppe – damit steuern Sie als Entwickler, in welcher Aktivierungsgruppe das Programm auszuführen ist.
- Angabe des Sonderwerts *NEW
  Damit legen Sie fest, dass das Programm in einer neuen Aktivierungsgruppe auszuführen ist. Das System übernimmt die Steuerung dieser neuen Aktivierungsgruppe.

Folgende Abbildung zeigt die Zuordnung der Programmkomponenten zu unterschiedlichen Aktivierungsgruppen.



*Aktivierungsgruppe *NEW*

3.2.4
Seite 22

Es ist zu beachten, dass die Angabe der Aktivierungsgruppe nur in Verbindung mit dem Kompilierbefehl CRTBNDPRG möglich ist. Hingegen können Aktivierungsgruppen nicht in Verbindung mit RPG-Modulen eingesetzt werden.

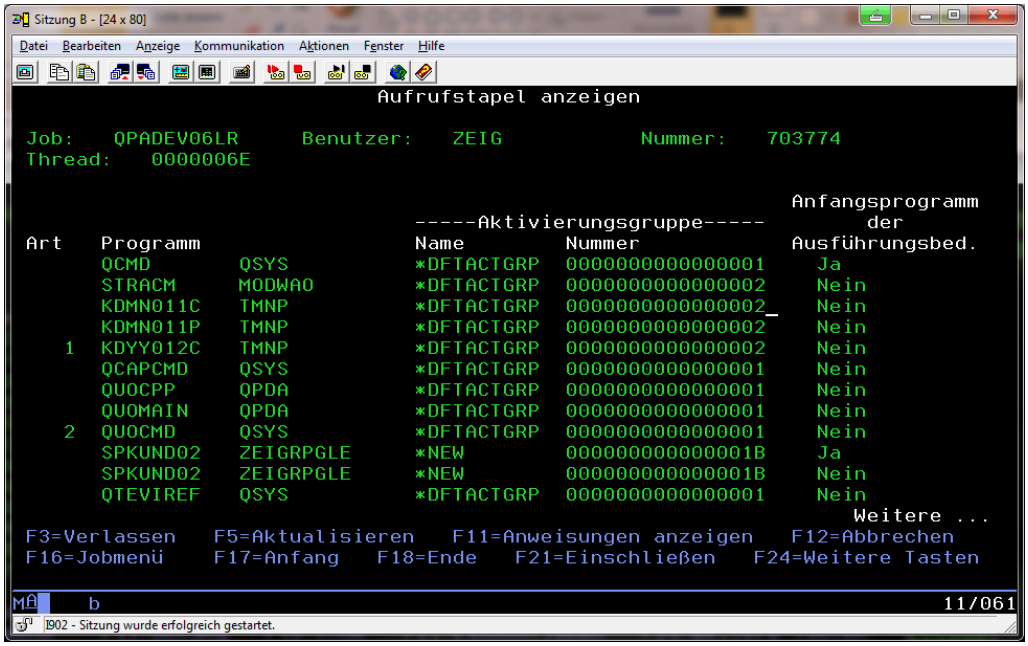
Zudem kann die ACTGRP-Angabe nicht in Verbindung mit DFTACTGRP(*YES) codiert werden. Diese beiden Angaben wären widersprüchlich und sind folglich unzulässig.

DFTACTGRP(*YES | *NO)
In direktem Zusammenhang mit dem Schlüsselwort ACTGRP steht das Schlüsselwort DFTACTGRP, welches festlegt, ob bei der Programmausführung die Standardaktivierungsgruppe zu verwenden ist. Wenn für dieses Schlüsselwort der Wert *NO angegeben wird, dann muss im Schlüsselwort ACTGRP die zu verwendende Aktivierungsgruppe hinterlegt sein. Nachfolgende Abbildung zeigt ein Beispiel hierfür.

```
h dftactgrp(*NO) actgrp(*CALLER)
```

Beispiel für DFTACTGRP

Bei der Angabe von *YES wird das RPG-Programm in der Standardaktivierungsgruppe ausgeführt.



Aktivierungsgruppen eines RPG-Programms

BNDDIR('binding-directory-name' {:'binding-directory-name'...})
Mit der Einführung von ILE haben sich auch die Binderverzeichnisse auf dem System i etabliert. Die Zuordnung eines Programmcodes zu einem Binderverzeichnis kann auf unterschiedliche Arten erfolgen. Eine Form der Zuordnung basiert auf der Steuerbestimmung BNDDIR, wo die Binderverzeichnisse angegeben werden können, die für den RPG-Code genutzt werden sollen. Dabei ist es möglich, den qualifizierten Namen – also die Bibliothek/den Namen des Binderverzeichnisses – oder nur den Namen des Binderverzeichnisses anzugeben. Im letztgenannten Fall sucht das System das benannte Binderverzeichnis anhand der Bibliotheksliste (*LIBL).

Die Angabe des Schlüsselwortes BNDDIR kann sowohl für die Erstellung von Modulen als auch für die Erstellung von RPG-Programmen erfolgen.

Auch hier gelten die ILE-Vorschriften, womit u. a. erklärt ist, dass die Angabe eines Binderverzeichnisses nicht zusammen mit dem Schlüsselwort DFTACTGRP(*YES) erfolgen darf. Dazu später mehr.

Nachfolgend finden Sie ein Beispiel für die Angabe eines Binderverzeichnisses in den Steuerbestimmungen.

```
H OPTION(*NODEBUGIO:*SRCSTMT)
H TIMFMT(*HMS:)  bnddir('KURSRPG')
H dftactgrp(*no) actgrp(*caller)
```

Beispiel für BNDDIR

3.2.4
Seite 24

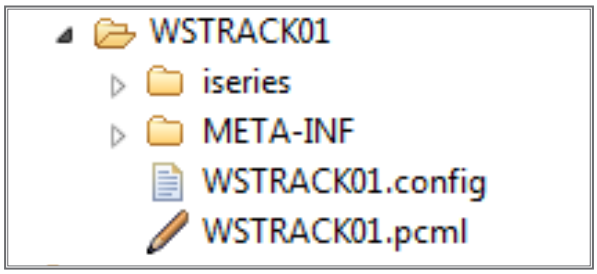
PGMINFO(*PCML | *NO { : *MODULE })
Jedes Programm verfügt über bestimmte Programm-Interface-Informationen (Schnittstelleninformationen), derer man sich u. a. auch beim Einsatz von RPG-Komponenten als WebServices bedienen kann. Es handelt sich dabei um eine großartige Möglichkeit, System-i-Anwendungen auch für Webanwendungen nutzbar zu machen. In diesem Zusammenhang hat IBM den Sonderwert *PCML für das Schlüsselwort PGMINFO eingeführt. PCML ist die Abkürzung für „Program Call Markup Language“ – ein Standard, der Aufrufe einheitlich beschreibt und interpretiert.

Wenn beispielsweise ein RPG-Modul zur Abfrage eines Lagerbestands erstellt werden soll, das auch von einem Webshop aus aufgerufen werden kann, dann kann dazu die Steuerangabe PGMINFO(*PCML:*MODULE) hinterlegt werden. Folgende Abbildung zeigt dafür ein Beispiel.

```
H decedit('0,')
H langid('DE')
H option(*SRCSTMT:*NODEBUGIO)
H PGMINFO(*PCML:*Module)
```

Beispiel für PGMINFO

Nur durch diese einfache Angabe kann dem Kompiler mitgeteilt werden, dass zusätzlich zum gewünschten Objekt (Modul oder Programm) PCML-Informationen zu generieren sind, die den WebService-Funktionen als Schnittstelle dienen. Die PCML-Informationen werden mit dem Kompilieren ins IFS gestellt und stehen z. B. auch nach der WebService-Implementierung im IFS bereit.



PCML-Beispiel im IFS

Free-Format-Steuerbestimmungen

Nachdem wir uns in den vorherigen Beispielen mit den Header-Bestimmungen in der Fix-Format-RPG-Version und der ersten Free-Format-Version beschäftigt haben, geht es im Weiteren um die finale und wirkliche Free-Format-Version.

Denn mit dieser stehen neben den Rechenbestimmungen im freien Format auch die Steuerbestimmungen in einer reinen Free-Format-Variante zur Verfügung. Dieser „neuen" Version können sich IBM-i-Betreiber seit der Version 7.1 mit dem TR 7 bedienen, und sie wird mit der Rational-Version 9.0.1 unterstützt.

Gemäß den Free-Format-RPG-Regeln ist jede Zeile im Free-Format-RPG-Code mit einem Semikolon zu beenden. So auch die Steuerbestimmungen.

Während die Steuerbestimmungen im Fix-Format RPG mit einem H an der Stelle 6 gekennzeichnet werden, ist in der Free-Format-Variante die Angabe CTL-OPT zu verwenden. Der Inhalt der Steuerbestimmungen in Form der Schlüsselwörter ändert sich nicht – lediglich die Einleitung für die Kennzeichnung einer Steuerbestimmungszeile!

Im Folgenden sind die Steuerbestimmungen in der Fix-Format- und in der Free-Format-Version gegenübergestellt. Zunächst die Steuerbestimmungen im Fix-Format.

```
.....  Free-Form++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     h bnddir('ZEIGBND' : 'QC2LE') dftactgrp(*No) actgrp('QILE')
     h option(*srcstmt : *nodebugio)
```

Steuerbestimmungen im Fix-Format

Hier folgt die Variante in der Free-Format-RPG-Version (ab TR 7!).

```
 Free-Form++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ctl-opt bnddir('ZEIGBND' : 'QC2LE') dftactgrp(*no) actgrp('QILE')
 option(*srcstmt : *nodebugio);
```

Steuerbestimmung im Free-Format

Obwohl es zwei Zeilen mit Steuerbestimmungen gibt, wurde nur in der ersten Zeile die Startangabe für die Steuerbestimmungen (CTL-OPT) codiert. Wie auch in den Free-Format-Rechenbestimmungen, kann bei den Steuerbestimmungen die Eingabe über eine Codezeile hinweggehen. In der finalen Zeile muss dann die Eingabe mit einem Semikolon beendet werden, damit die Editoren und Kompiler wissen, dass das Ende der Zeile erreicht ist.

3.2.4
Seite 26

Diese beispielhaft aufgezeigten Schlüsselwörter für die Steuerbestimmungen sollen exemplarisch einen Eindruck in die Möglichkeiten vermitteln, die Steuerbestimmungen bieten. Während einige Steuerbestimmungen in den letzten Jahren durch die Verfügbarkeit der Rational-Werkzeuge an Bedeutung verloren haben, sind andere gerade in den jüngsten RPG-Versionen ein beinahe tägliches Werkzeug der Anwendungsentwickler.

3.2.4.3 RPG-Dateibestimmungen

RPG hat – wie jede andere Programmiersprache auch – ihre Besonderheiten. Beispielsweise muss im RPG-Programm für jede Datei, die in einem Programm verarbeitet werden soll, eine Beschreibung hinterlegt werden, die darauf verweist, mit welcher Datei in welcher Form gearbeitet werden soll. Für jede Datei, die nativ im RPG-Programm genutzt werden soll, müssen zu Beginn des Programmcodes Angaben getätigt werden, die über die folgenden Mindestinformationen verfügen:

- den Dateinamen,
- die Verarbeitungsart (ob nur lesend oder ob bereits bestehende Datensätze verändert werden),
- ob neue Elemente in die Datei geschrieben/hinzugefügt werden,
- die Organisationsart der Datei,
- den Dateityp (ob es sich um eine Datenbankdatei, die letztlich Daten verarbeitet, handelt; ob eine Bildschirm- oder auch eine Druckerdatei definiert wird).

Wenn Dateiangaben in einem RPG-Programm definiert werden, dann handelt es sich um Rechen- oder F-Bestimmungen (F = File).

Es ist immens wichtig, dass beim Codieren eines RPG-Programms die Reihenfolge der Anweisungen beachtet wird, z. B. , dass nach den optionalen H-Bestimmungen oder Steueranweisungen die Dateibestimmungen in Form der F-Bestimmungen codiert werden.

Nachfolgende Abbildung zeigt exemplarisch die Angabe von Dateien in einem RPG-Programmcode.

```
002700     h actgrp('VERTRIEB')
002800     h*dftactgrp(*NO) actgrp(*CALLER)
002900     h copyright('Joerg Zeig 2015')
003000
003100      //-------------------------------------------------------
003200      //    D A T E I - D E F I N I T I O N E N
003300      //-------------------------------------------------------
003400
003500      //
003600     fTEIL01    if   e           k disk
003700      //
003800     fTEOS01    if   e           k disk
003900      //
004000     fLfsExErr  if a e           k disk
004100      // LFS Teilestamm
004200     fPANfKOP2  uf A e           k disk
004300
```

RPG-Codebeispiel mit Dateiangaben

3.2.4
Seite 28

Die Angabe der Rechenbestimmungen ist nur dann notwendig, wenn in einem RPG-Programm auch tatsächlich mit Dateien gearbeitet wird, was nicht immer der Fall ist. Auch wenn das originäre Ziel eines Programms meist die Verarbeitung von Daten ist, kann es in der Praxis durchaus auch RPG-Programme oder RPG-Module geben, die nicht direkt mit Dateien arbeiten. Aus diesem Grund ist die Angabe der Dateibestimmungen nicht zwingend erforderlich, sondern richtet sich vielmehr nach den Anforderungen des einzelnen Programms.

Anmerkung:

Es besteht zwar noch immer die Möglichkeit, Dateidefinitionen intern im RPG-Programm zu hinterlegen, doch sollte gemäß den Empfehlungen der IBM auf diese antiquierte Form der Dateidefinition verzichtet werden. Folglich sollten im Regelfall nur solche Dateien in einem RPG-Programm definiert werden, die extern beschrieben worden sind. Insbesondere bei der Verwendung von Datendateien kann zudem zwischen den extern beschriebenen ISAM-Dateien (physische und logische Dateien) und den SQL-beschriebenen Datenbankelementen (Tabellen, Ansichten) gewählt werden. Aus Sicht von RPG werden diese beiden Varianten gleichartig definiert, was bedeutet, dass z. B. eine physische Datei im selben Format definiert werden kann wie eine vergleichbare SQL-Tabelle.

SQL-beschriebene Elemente werden in RPG analog zu deren ISAM-Gegenstücken verwendet, solange kein embedded SQL genutzt wird. Sollte jedoch diese Version der RPG-Programmierung zur Anwendung kommen, dann kann auf Dateibestimmungen verzichtet werden. Müssen Dateiinhalte nativ verarbeitet werden (mit klassischen RPG-Befehlen und nicht mittels SQL), dann werden die F-Bestimmungen zu Beginn des Programmcodes nach den optionalen H-Anweisungen codiert.

Bei den Rechenbestimmungen muss bis zum Erscheinen der Betriebssystemversion 7.1 unbedingt das fixe Format verwendet werden. So lange besteht also noch eine Bindung an die spaltenorientierte Version mit Beachtung und Einhaltung der vorgegebenen Spalten. Erst mit den neueren Release-Versionen sind in der finalen Free-Format-RPG-Variante die Dateibestimmungen in einem freien Format eingebbar.

Gerade die Dateibestimmungen spiegeln in ihrer Komplexität die unterschiedlichen RPG-Versionen und die damit zur Verfügung stehenden Programmier- und Datenbanktechniken wider. Wenn heute ein neues und modernes RPG-Programm geschrieben werden soll, dann ist es wichtig, moderne Wege einzuschlagen. Leider kann an dieser einführenden Stelle nicht ausgiebig genug auf die RPG-Klassik und die RPG-Moderne eingegangen werden, aber dennoch sei hier erwähnt, dass einige Eingabebereiche in den Dateibestimmungen heutzutage nicht mehr verwendet werden. Bei der Auflistung der Spalten wird bei denen, die heute nicht mehr genutzt werden, der entsprechende Hinweis gege-

ben. Die hier zitierte Klassik und Moderne haben nichts mit der Editorform zu tun, sind also nicht auf Free-Format RPG bezogen. Es ergeben sich gerade in der wirklichen Free-Format-Version, die sich eben nicht nur auf die Rechenbestimmungen, sondern auch auf die Dateibestimmungen bezieht, wesentliche Änderungen. Dazu später mehr.

Da die Vielzahl der RPG-Codes in existierenden Umgebungen in Fix-Format RPG oder der ersten Free-Format-Variante geschrieben sind, erfolgt zunächst eine Beschäftigung mit der spaltenorientierten Version.

Erläuterungen zum Einsatz von Dateien in RPG:

Stellt man sich die Frage, was eigentlich eine Datei ist, dann fällt hinsichtlich IBM i auf, dass dort nicht nur Dateien vorzufinden sind, die in direktem Zusammenhang mit den hiesigen Dateien stehen, sondern dass vielmehr im Betriebssystem IBM i Dateien auch dazu genutzt worden sind, um das Erscheinungsbild von Bildschirmanzeigen im 5250-Umfeld definieren und Druckausgaben beschreiben zu können. Folgende Dateien stehen beispielsweise für ein RPG-Programm zur Verfügung::

- Dateien im klassischen Sinn für die Speicherung und Bereitstellung von Daten im ISAM DB2-Konzept:
  - Physische Dateien
  - Logische Dateien
  - Join-Dateien

  Diese Art Dateien ist in den Dateibestimmungen als Einheit „DISK“ zu hinterlegen.
- „Dateien“ im SQL-Umfeld für die Speicherung und Bereitstellung von Daten im SQL-Konzept:
  - Tabellen
  - Ansichten
  - Indizes

  Diese Art Dateien ist in den Dateibestimmungen als Einheit „DISK“ zu hinterlegen.

  Die Eingabe der SQL-Datenbankelemente erfolgt in derselben Weise wie auch die für die klassisch beschriebenen ISAM-Dateien – zumindest so lange, wie diese mit nativen RPG-Befehlen und nicht mit embedded SQL verarbeitet werden.

3.2.4
Seite 30

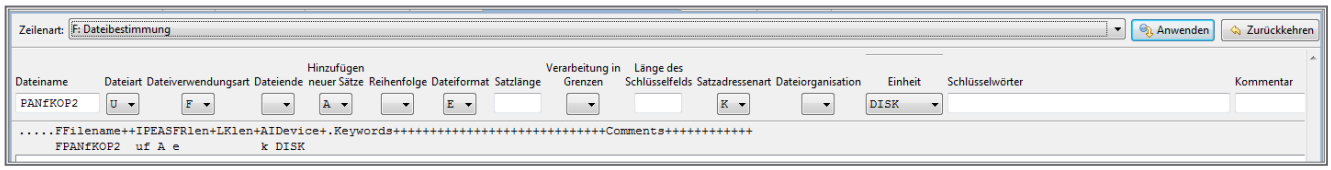
- Workstation-Dateien (DSPF)

  Diese Dateien beschreiben das Aussehen und das Verhalten der 5250-Green-Screen-Oberflächen. Es handelt sich um eine besondere Dateiform, die in den Dateibestimmungen mit der Einheitenangabe WORKSTN codiert werden muss.

- Druckerdateien (PRTF)

  Diese Dateien beschreiben das Aussehen und das Verhalten von Druckerdateien. Es handelt sich um eine besondere Dateiform (PRTF), die mit der Einheitenangabe PRINTER codiert werden muss.

Nachfolgend sehen Sie den Aufbau der Dateibestimmungen in einer Prompt-Anzeige.



Bedienerführung für eine Dateibestimmung

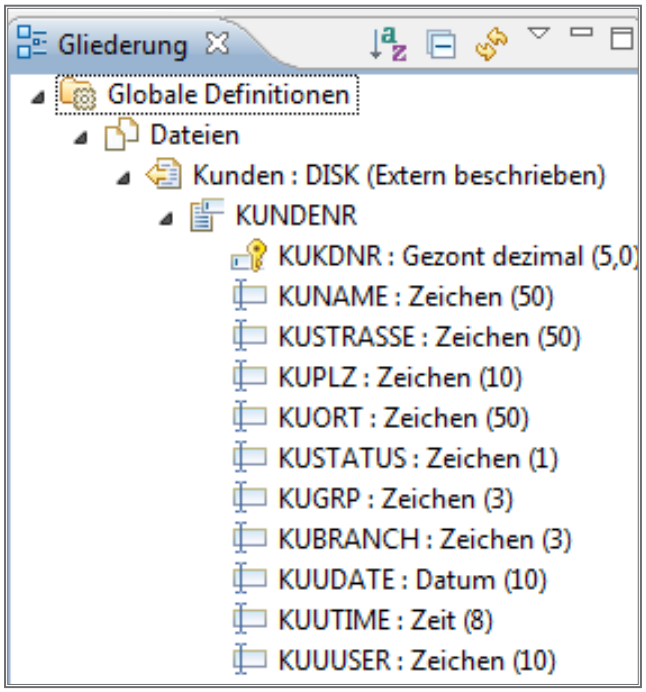
Welche Bedeutung haben nun die einzelnen Spalten?

Wie bei den übrigen RPG-Bestimmungen auch, so gilt für die Dateibestimmungen, dass deren Bereich in Spalte 6 beginnt. Dort wird für eine Dateibestimmung „F“ (File) angegeben.

Dateiname (Spalten 7 – 16)

Hier ist der Name der Datei anzugeben, wobei sichergestellt sein muss, dass er eindeutig ist. Wenn Sie eine extern beschriebene Datei angeben wollen, dann mit deren Namen ohne Qualifizierung. Bei Bedarf kann der interne Name einer Datei im RPG überschrieben werden. Das kann etwa dann Sinn machen, wenn zwei gleichnamige Dateien in einem RPG-Programm verarbeitet werden sollen (z. B. eine Datei UMSATZ in der Echtbibliothek und eine weitere Datei UMSATZ in einer Testbibliothek). In diesem Fall kann der Dateiname in Verbindung mit dem Schlüsselwort EXTDESC angegeben werden. Dazu später mehr.

Mit Angabe des Dateinamens und der Verwendung wird später beim Kompilieren der Inhalt auf Datei-, Satzformat- und Feldebene im RPG zur Verfügung stehen. Wenn Sie als Editor den LPEX-Editor aus den Rational-Produkten verwenden, dann sehen Sie die eingebundenen Informationen dort sehr schön in der Gliederung. In den globalen Definitionen werden auf Basis der in den Dateibestimmungen angegebenen Dateinamen die Details ausgelesen, und sie stehen somit dem Programm und dem Anwendungsentwickler zur Nutzung bereit. Folgende Abbildung zeigt exemplarisch die Dateiinformationen, die im Rational Developer visualisiert werden.



Dateidarstellung im LPEX-Editor (Gliederung)

Dateiart (Spalte 17)

Mit diesem Parameter wird festgelegt, wie die Datei vom RPG-Programm zu öffnen ist. Eine wesentliche Rolle spielt dabei die Form der Datenverarbeitung. Mögliche Werte sind:

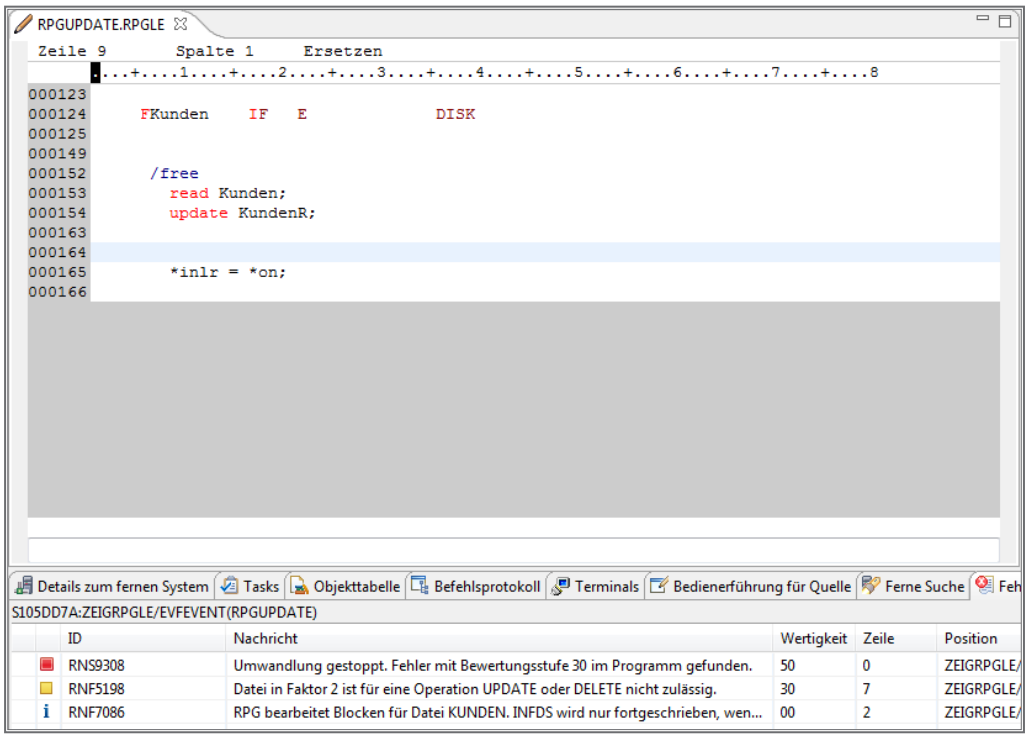
- I = Input
  Diese Einstellung ist dann zu verwenden, wenn auf den Inhalt einer Datei nur lesend zugegriffen wird. Damit sind keine Datenmanipulationen möglich.
- O = Output
  Mit dieser Einstellung wird festgelegt, dass Datensätze in die Datei ausgegeben werden.
- U = Update
  Mit dieser Einstellung kann festgelegt werden, dass der Inhalt einer Datei sowohl gelesen als auch fortgeschrieben, also verändert werden kann. Auch Löschungen müssen über die Dateiart „U“ durchgeführt werden.
- C = Combined Files
  Diese Angabe definiert eine Datei als Eingabe- und auch als Ausgabedatei und kommt meist in Verbindung mit WORKSTN-Dateien zum Einsatz – also bei der Definition von Bildschirmdateien und weniger bei Datendateien.

3.2.4
Seite 32

Die Dateiart legt u. a. fest, wie der Inhalt der angegebenen Datei verarbeitet werden soll. Beim Erstellen von RPG-Programmen und deren Ausführung ist es wichtig zu bedenken, dass eine Datei erst nach initialem Öffnen vom RPG-Programm genutzt werden kann. Das Öffnen kann dabei wahlweise automatisch vom RPG-Programm während der Ausführung der jedem RPG-Programm eigenen Initialisierungssubroutine erfolgen oder programmgesteuert durch entsprechende Befehle eingeleitet werden. Es gibt Anforderungen, bei denen das Ziel nur dann erreicht werden kann, wenn der Anwendungsentwickler gezielt das Öffnen einer Datei durchführt. Dazu später mehr. Zuvorderst soll davon ausgegangen werden, dass Dateien, die im hiesigen RPG-Programm genutzt werden, automatisch über einen Programminitialisierungslauf geöffnet werden.

Folgendes Codebeispiel verarbeitet die Datei „Kunden“. Diese Datei ist als Eingabedatei (I) in der Spalte 17 definiert, und damit wird die Datei beim Programmaufruf zum Verarbeiten des Inhalts als Lesezugriff geöffnet. Folglich können später im Programmablauf alle Anweisungen genutzt werden, mit denen lesend auf den Dateiinhalt zugegriffen wird.

Wird indes eine Anweisung verwendet, die Daten manipulieren soll, dann prüft dies der Compiler; er quittiert die Unstimmigkeit mit einer Fehlermeldung RNF5198, die besagt, dass ein Fortschreiben des Inhalts der Datei so nicht möglich ist.



Fehlermeldung-Update auf i-Datei nicht möglich

Dateiverwendung (Spalte 18)

Gültige Werte zur Dateiverwendung sind:

- P = Primärdatei
  Bei einer solchen Datei werden die Datensätze in der Datei vom Beginn bis zum Ende über den RPG-Zyklus verarbeitet. Das bedeutet, dass das folgende Verfahren Öffnen / Lesen / Verarbeiten / Schließen verwendet wird. In einem RPG-Programm kann es maximal eine Primärdatei geben.
- S = Sekundärdatei
  Diese Angabe wird dann getätigt, wenn in einem RPG-Programm mehrere Dateien verarbeitet werden sollen. Diese Form wird in Verbindung mit der Primärdatei verwendet.
- R = Record-Address-Datei
- T = Table oder Array
- F = Fullprocedural
  Die Verarbeitungsreihenfolge wird vom Anwendungsentwickler festgelegt.

Anmerkung:

RPG kann in unterschiedlichen Versionen genutzt werden. In modernen Ansätzen verzichtet man als guter Entwickler auf die Verwendung von Primär- und Sekundärdateien. Für Datendateien an dieser Stelle ist die Einstellung F = Vollprozedural vorzüglich. Überlegen Sie sich bitte gut, ob Sie in neuen Programmen wirklich Primär- und Sekundärdateien verarbeiten wollen!

Dateiende (Spalte 19)

Mit diesem Wert wird festgelegt, wie Primär- und Sekundärdateien verarbeitet werden sollen und wie das Ende im Programm erreicht wird. Wird in dieser Spalte ein „E“ eingetragen, dann müssen im RPG-Zyklus alle Datensätze einer Datei verarbeitet werden, bis mit dem RPG-Ende-Schalter „LR“ das Programmende gesetzt wird. Diese klassische Form der RPG-Verarbeitung sollte jedoch in modernen Programmen nicht mehr verwendet werden, und es sollte vielmehr diese Spalte leer gelassen werden.

In neueren Programmcodes, die nicht über den RPG-Zyklus alleine gesteuert werden, kann der Wert dieser Spalte leer bleiben (siehe auch das RPG-Codebeispiel).

3.2.4
Seite 34

Hinzufügen neuer Sätze (Spalte 20)

Leider gibt es vonseiten der IBM keine einheitliche Linie, wenn es um das Hinzufügen neuer Datensätze in eine Datei geht. Sollten Sie eine Datei in der Definition der Dateiart (Spalte 17) als Ausgabedatei (O) definiert haben, dann kann der Wert dieser Spalte leer bleiben. Es kann allerdings für Eingabedateien (I) oder Update-Dateien (U) angegeben werden, dass bei diesen Dateiarten auch neue Datensätze in die angegebene Datei geschrieben werden dürfen. Dann muss in dieser Spalte der Wert „A“ (Add) eingetragen werden. Anderenfalls würden Schreibprozesse für neue Datensätze (Anweisung WRITE) fehlschlagen, bzw. der Compiler würde bereits eine Diskrepanz in der Definition feststellen und kein ausführbares Programmobjekt erstellen. Dieser Wert kann nur in Verbindung mit DISK-Dateien gesetzt werden.

Reihenfolge (Spalte 21)

Auch diese Spalte hat nur dann eine Auswirkung, wenn es sich um eine Primär- oder Sekundärdatei handelt. Wie bereits erläutert, sollten Sie auf diese klassische Definitionsart der Dateien verzichten. Sollten Sie jedoch in älteren Programmcodes diesen Eintrag finden, dann hat er folgende Bedeutung:

- Leer
  Standardeinstellung für alle Dateien, die nicht als Primär- oder Sekundärdatei definiert sind
- A = Ascending (aufsteigend)
  Die angegebenen Felder werden in aufsteigender Reihenfolge sortiert.
- D = Descending (absteigend)
  Die angegebenen Felder werden in absteigender Reihenfolge sortiert.

Dateiformat (Spalte 22)

Dateien können, je nachdem, aus welcher Entwicklungsära sie stammen, als programmbeschriebene oder als „interne“ Dateien definiert oder als extern beschrieben genutzt werden. Bei den extern beschriebenen Dateien nutzt der RPG-Compiler die Dateidefinition, die im Dateiobjekt liegt, und stellt die Inhalte (Satzformat, Felder etc.) für die Verwendung im RPG-Programm zur Verfügung.

Verzichten Sie auch hier auf die Möglichkeit, Dateien in einem Programm zu beschreiben. Diese wirklich altmodische Form der Dateibeschreibung sollte nur noch in bestehenden Programmen vorzufinden sein, nicht aber in neuen. Verzichten Sie also auf die Definition von intern beschriebenen Dateien (Angabe „F“) und nutzen Sie extern beschriebene Dateien (Angabe „E“).

Satzlänge (Spalten 23 – 27)

Mit diesem Wert wird die Satzlänge jedes einzelnen Datensatzes angegeben. Im Regelfall wird für moderne RPG-Programme der Standardwert blank verwendet; es wird, also in diesen Spalten nichts eingegeben. Auch diese Spaltenangaben sind mehr Relikte aus den Anfängen von RPG und beziehen sich auf programmbeschriebene Dateien.

Für neue RPG-Programme sollte der Standardwert leer sein. Es sollte also keine Eingabe erfolgen.

Verarbeitung in Grenzen (Spalte 28)

Mit dieser Spalte wird angegeben, ob es Limitierungen bei der Verarbeitung der Datei gibt. Auch dieser Spaltenwert ist antiquiert und sollte in einem modernen RPG-Programmcode nicht mehr auftauchen.

ARTLMOD03.RPGLE | LIMITRPG01.RPGLE

```
Zeile 25      Spalte 67    Ersetzen
     .........1....+....2....+....3....+....4....+....5....+....6....+....7....+....
000102       //*****************************************************************
000103       //
000104       // Verwendung von Limit Angaben auf Dateiebene
000105       // LIMITRPG01
000106       //
000107       // Copyright (c) J.Zeig / ZeBIS 2013
000108       // Hier werden nur die Datensätze der Datei verarbeitet, deren
000109       // Nummern innerhalb der angegeben Grenzen liegen
000120      FLIMITS    IR   F    6     3  DISK    RAFDATA(KUNDEN)
000121      FKUNDEN    IP   F   28L    3PIDISK    KEYLOC(1)
000122      FPRINT     O    F   80        PRINTER
```

Beispiel für die Verarbeitung in Grenzen

Länge des Schlüsselfeldes (Spalten 29 – 33)

Diese Spalte findet nur für programmbeschriebene Dateien Verwendung und muss für extern beschriebene Dateien leer bleiben.

3.2.4
Seite 36

Satzadressenart (Spalte 34)

Die Verarbeitung eines Dateiinhalts geschieht in einer bestimmten Reihenfolge. In dieser Spalte sind folgende Eingaben möglich:

- Leer
  Für die Verarbeitung werden die relativen Satznummern verwendet und keine definierten Schlüssel genutzt, was durchaus gewünscht sein kann. Wenn die Dateiinhalte jedoch in einer definierten Reihenfolge verarbeitet werden sollen, dann muss hier ein Eintrag erfolgen.
- A (Zeichenfelder / Character)
  Nur gültig für programmbeschriebene Dateien
- P (gepackte Schlüssel)
  Ebenfalls nur für programmbeschriebene Dateien zu verwenden
- K (Schlüsselwerte)
  Nur in Verbindung mit extern beschriebenen Dateien zu verwenden. Wenn eine Datei mit Schlüsseln verwendet werden soll, dann wird hier ein „K“ eingegeben.

Dateiorganisation (Spalte 35)

Sie ahnen es vielleicht: Auch dieser Wert kommt in Verbindung mit programmbeschriebenen Dateien zum Einsatz und sollte, wie viele andere Spaltenwerte auch, heute nicht mehr genutzt werden.

Einheit (Spalten 36 – 42)

In den Dateibestimmungen muss dem RPG-Programm mitgeteilt werden, um welchen Datentyp es sich handelt, und zwar um eine Datei, die Daten beinhaltet oder bereitstellt (DISK), oder etwa um eine Anzeige- bzw. Druckerdatei. Über die Einheit werden die im weiteren RPG-Verlauf möglichen Funktionen und Arbeiten festgelegt, die in Verbindung mit der Datei und der Einheit erlaubt sind.

Gültige Werte sind:

- DISK
  Dabei handelt es sich um eine Datei, die Daten beinhaltet. Dieser Wert ist für alle physischen und logischen Dateien sowie die jeweiligen SQL-Gegenstücke zu hinterlegen. Bei Bedarf kann hier auch eine DDM-Datei mit der Angabe DISK genutzt werden.
- PRINTER
  Wenn Daten zu Papier oder in eine Druckerdatei gestellt werden sollen, dann muss hier der Wert „PRINTER“ angegeben werden.

- WORKSTN
  Auch Bildschirmdateien (5250-Dateien) sind eine besondere Form von Dateien, die natürlich im RPG entsprechend individuell zu behandeln sind. Wenn also mit einer DSPF (Display File) gearbeitet werden soll, dann muss dazu im Bereich „Einheit“ der Wert „WORKSTN“ angegeben werden.

Nachfolgendes Codebeispiel zeigt exemplarisch die unterschiedlichen Einheiten für Datendateien, Druckerdateien und Anzeigedateien.

```
h dftactgrp(*NO) actgrp(*CALLER)
h copyright('Joerg Zeig 10/2014')
FArtikel   if   e           K DISK
FDSPSQL02  CF   E             WORKSTN InDDS(WkStnInd)
FPRTSQL02  O    E             PRINTER Oflind(NextPage)
```

Beispiel für unterschiedliche Einheitenangaben

Schlüsselwörter (Spalten 44 – 80)

Die Schlüsselwörter kommen in Verbindung mit vielen Konstellationen vor und erlauben zusätzliche Steuerungen und Definitionen. Im vorhergehenden Codebeispiel sahen Sie einige Schlüsselwörter, deren Einsatz in Kombination mit verschiedenen anderen Spalten der Dateibestimmungen möglich ist. Auf einige der möglichen Schlüsselwerte soll hier eingegangen werden, eine umfassende Auflistung findet sich jedoch in den Referenzen der IBM.

Verschiedene Schlüsselwörter sind alleine nutzbar, andere wiederum können nur zusammen mit einem weiteren Schlüsselwort definiert werden. Die Schreibweise (Klein-/Großschreibung) ist bei der Codeeingabe des Schlüsselwortes selbst unerheblich. Allerdings sollten Sie darauf achten, dass Bestandteile des Schlüsselwortes, wie z. B. Konstanten, korrekt eingegeben werden. So kann es z. B. bei der qualifizierten Angabe von Dateinamen zu Fehlern kommen, wenn diese nicht durchgängig in Großbuchstaben codiert wurden. Dazu später mehr.

Wenn mehrere Schlüsselwörter für eine Datei anzugeben sind, dann können diese in den Folgezeilen unmittelbar hinter der ersten Dateibestimmung codiert werden. Bei Bedarf lassen sich hier auch Kommentarzeilen einfügen. Allerdings darf die Kette der zu einer Datei gehörigen Zeilen nicht durch andere Programmcodezeilen unterbrochen werden.

3.2.4
Seite 38

Im folgenden Codebeispiel finden Sie drei verschiedene Schlüsselwörter, die sich auf die Datei „Kunden“ beziehen.

```
FKunden    IF   E             DISK    Include(Format01)
F                                     Prefix('X_')
 * und hier noch die Commitsteuerung
F                                     commit
```

Beispiel für eine multiple Schlüsselwortangabe auf Dateibestimmungsebene

INDDS

Das Schlüsselwort INDDS kann für Druck- und Anzeigedateien verwendet werden. Es ist also nur für die Einheiten „PRINTER“ und „WORKSTN“ eingebbar. Mithilfe dieses Schlüsselwortes lassen sich Indikatoren bzw. Bezugszahlen so definieren, dass sie leichter zu verarbeiten und zu interpretieren sind. Das Schlüsselwort INDDS muss immer zusammen mit dem Namen der zu verwendenden Datenstruktur definiert werden, die in Klammern hinter dem Schlüsselwort anzugeben ist und im weiteren Verlauf des RPG-Programms beschrieben sein muss. Im folgenden Beispiel wird die Bezugszahl 03 im RPG-Programm mit dem Wert „Exit“ quasi übersteuert. An dieser Stelle ist anzumerken, dass in einer Anzeige- und einer Druckerdatei keine benannten Indikatoren verwendet werden dürfen. Das heißt, Sie sind auf die Verwendung von Bezugszahlen, die sich in sogenannte benannte Indikatoren umwandeln lassen, angewiesen. Mit dieser Technik können aus abstrakten Bezugszahlen leichter erkennbare benannte Indikatoren gemacht werden.

```
FArtikel   if   e           K DISK
FDSPSQL02  CF   E             WORKSTN InDDS(WkStnInd)
FPRTSQL02  O    E             PRINTER Oflind(NextPage)
D WkStnInd        DS
D  Exit                   3       3N
```

Beispiel für INDDS in Verbindung mit der Datenstruktur WKStnInd

Wenn Sie INDDS verwenden, dann müssen Sie darauf achten, dass folgende Bedingungen eingehalten werden:

- Die interne Länge der Datenstruktur ist immer 99 und entspricht damit der Anzahl der verwendbaren Bezugszahlen in WORKSTN- und PRINTER-Dateien.
- Die Initialisierung erfolgt mit 0.
- Die Datenstruktur muss im RPG-Programm definiert werden (in den Definitionsbestimmungen „D“).
- Eine Datenstruktur kann in Verbindung mit mehreren Dateien eingesetzt werden.

OfLind

Die Steuerung der Druckerdateien in einem RPG-Programm unterliegt ebenfalls bestimmten Regeln. Wie bei den Anzeigedateien auch, so kann der Anwendungsentwickler in Druckerdateien über Bezugszahlen eine gewisse Form der Steuerung abbilden. Ein Indikator, der häufig für Druckerdateien genutzt wird, ist das Kennzeichen, mit dem der Seitenüberlauf gesteuert wird. Dieser Indikator zeigt dem Programm also an, wann eine neue Seite begonnen werden muss.

Das Schlüsselwort OfLind ist nur im Zusammenhang mit einer PRINTER-Datei eingebbar.

Gültige Werte sind:

- Name (siehe nachfolgendes Beispiel)
  Angabe des Namens für den Indikator. Dabei muss es sich um ein Feld handeln. Hier ist keine Angabe einer Datenstruktur möglich.
- *INOA – *INOG
  Diese fest zuordenbaren Bezugszahlen für die Steuerung von Überlaufanzeigern lassen sich in Verbindung mit OfLind angeben.
- *IN01 – *IN99
  Diese klassische Variante der Bezugszahl wird ebenfalls für das Schlüsselwort OfLind unterstützt.

Nachfolgend sehen Sie ein Beispiel für OfLind zusammen mit der Definition des Überlaufanzeigers „Nextpage“.

```
FArtikel   if   e           K DISK
FDSPSQL02  CF   E             WORKSTN InDDS(WkStnInd)
FPRTSQL02  O    E             PRINTER Oflind(NextPage)
D WkStnInd        DS
D  Exit                    3       3N
D* Anzeiger fuer den Seitenwechsel
DNextPage         S                N
```

OfLind zusammen mit der Definition des Indikators

3.2.4
Seite 40

ExtDesc

Mit diesem Schlüsselwort wird dem Umwandlungsprozess mitgeteilt, welche Datei zum Wandlungszeitpunkt verwendet werden soll. Die damit hinterlegten Dateiinformationen werden für das Programm eingelesen und für die weitere Verarbeitung genutzt. Wichtig dabei ist zu beachten, dass das Schlüsselwort ExtDesc nur zum Zeitpunkt des Kompilierens Anwendung findet und sich nicht auf den Zeitpunkt der Programmausführung auswirkt.

Die Angabe des Dateinamens kann in beliebiger Weise erfolgen. So können Sie z. B. den einfachen Dateinamen angeben (in diesem Fall wird anhand der Bibliothekssuchliste die zuerst gefundene Datei genutzt), oder Sie können den qualifizierten Dateinamen eingeben (das ist der Bibliotheksname, in dem sich die betreffende Datei befindet, gefolgt von einem Schrägstrich und anschließend dem Dateinamen). Bei der Angabe des qualifizierten Dateinamens kann auch der Sonderwert *LIBL angegeben werden, was die Suche nach den ersten Dateien der Suchliste wiederum zur Folge hat. Der Sonderwert *CURLIB ist an dieser Stelle nicht gültig!

Anmerkung:

Da die Angabe des Dateinamens in Hochkommazeichen erfolgen muss, muss dieser unbedingt in Versalien geschrieben werden. Anderenfalls wird die Datei trotz mutmaßlich korrekter Schreibweise nicht gefunden.

Nachfolgendes Codebeispiel zeigt die Verwendung von ExtDesc, zuerst als Alleinangabe und dann zusammen mit dem Schlüsselwort ExtFile.

```
 // zum Kompilieren wird die Datei ZEIGRPG/KUNDEN verwendet
 // Damit werden die Definition für file1 belegt
 // ohne EXTFILE wird zur Ausführung die Datei KUNDEN über *LIBL
 // gesucht
Ffile1     if   e             disk
F                                      extdesc('ZEIGRPG/KUNDEN')
 // zum Kompilieren wird die Datei ZEIGRPG/KUNDEN2 verwendet
 // Damit werden die Definition für file1 belegt
 // mit EXTFILE wird zur Ausführung die Datei KUNDEN über den
 // qulifizierten Namen (ZEIGRPGLE/KUNDEN2) gesucht
Ffile2     if   e             disk
F                                      extdesc('ZEIGRPG/KUNDEN2')
F                                      extfile(*extdesc)
```

Beispiele für ExtDesc

ExtFile

Mit dem Schlüsselwort ExtFile wird festgelegt, welche Dateien in welcher Bibliothek zu öffnen sind. Das Schlüsselwort ist ähnlich wie das Schlüsselwort ExtDesc mit dem einfachen oder dem qualifizierten Dateinamen anzugeben. Zusätzlich ist auf der Ebene des Schlüsselworts ExtFile der Sonderwert *EXTDESC definierbar. Dieser verweist auf den Inhalt, der im Schlüsselwort EXTDESC hinterlegt wurde. Ein Beispiel dafür zeigt die vorhergehende Abbildung. Wird der Sonderwert *EXTDESC eingegeben, dann muss zwingend auch eine EXTDESC-Angabe im Programmcode enthalten sein. Bei Bedarf kann für dieses Schlüsselwort anstelle des Dateinamens auch eine Variable verwendet werden. Wird dieses Verfahren angewandt, dann müssen Sie als Anwendungsentwickler sicherstellen, dass die in der Variablen enthaltene Datei existiert und dass der Variablen vor dem Öffnen der Datei der entsprechende Wert zugewiesen wurde.

Auch für dieses Schlüsselwort sehen Sie hier ein Codebeispiel.

```
 // Angabe Dateiname qualifiziert - auf Großschreibung achten!!!
Ffile1     IF   F   10        DISK    EXTFILE('ZEIGRPGLE/KUNDEN'

 // Angabe Dateiname einfach - auf Großschreibung achten!!!!
Ffile2     IF   F   10        DISK    EXTFILE('KUNDEN')

 // Angabe Dateiname als Variable!!!
Ffile3     IF   F   10        DISK    EXTFILE(kunden)

 // Angabe Dateiname als Variable mit benutzergesteuertem Öffner
Ffile4     IF   F   10        DISK    EXTFILE(Kunden)
F                                     USROPN
D kunden          S              21A
```

Beispiele für ExtFile

Beachten Sie in obigem Beispiel die verschiedenen Eingabeoptionen für das Schlüsselwort ExtFile. Weil die beiden ersten Varianten die Eingabe des fixen Dateinamens als Konstante zeigen, ist die Großschreibung innerhalb der Hochkommazeichen unbedingt einzuhalten.

Anmerkung:

Wie in anderen Bereichen von RPG auch, so wird auf der Basis der Dateibestimmungen die Konstantendefinition stets in Hochkommazeichen vorgenommen.

Die beiden letzten Varianten der ExtFile-Verwendung zeigen die Nutzung von Variablen. Die Unterscheidung der Angabe zu File3 und File4 liegt in einem weiteren Dateibestimmungsschlüsselwort – dem Schlüsselwort USROPN.

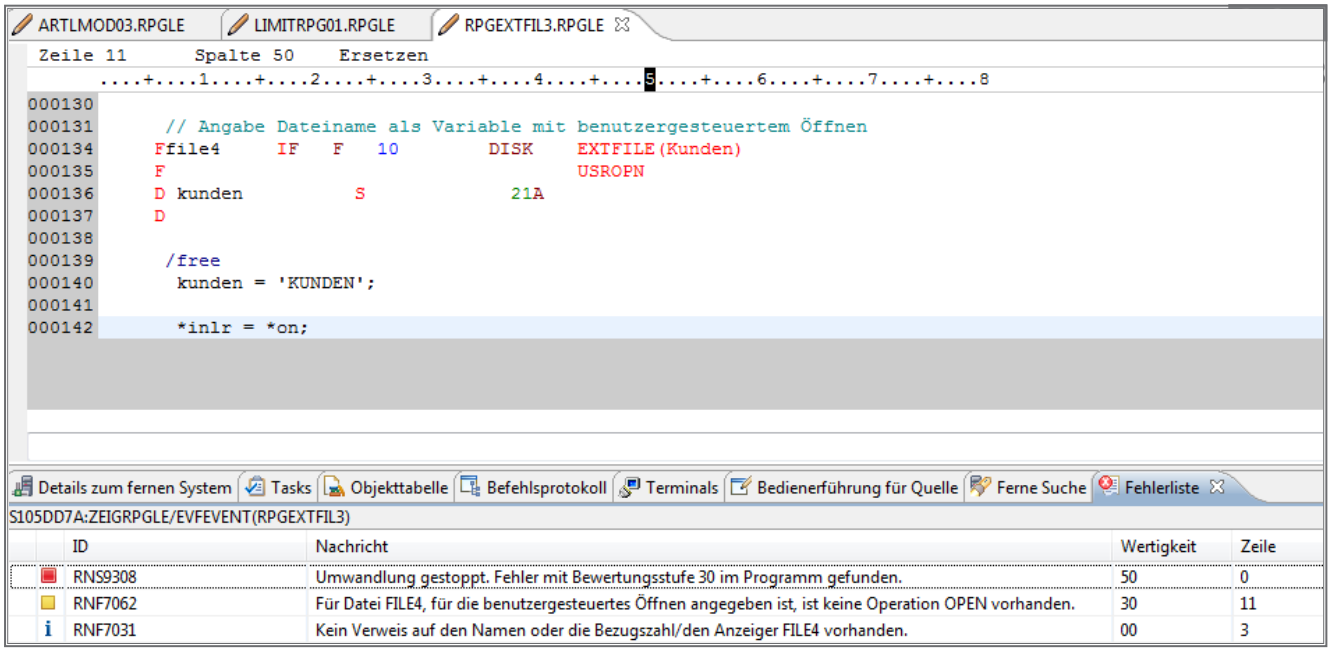
Zur Wiederholung: Jede Datei muss, damit sie in einem RPG-Programm genutzt werden kann, in einer Initialisierungsphase zunächst geöffnet werden. Im Beispiel File3 übernimmt das Programm das Öffnen der Datei. Im Beispiel File4 wird mit dem zusätzlichen Schlüsselwort USROPN festgelegt, dass die Datei korrekt geöffnet werden soll und damit für die weitere Verwendung im RPG-Ablauf zur Verfügung steht.

UsrOpn

Wie eingangs erwähnt, muss jede Datei beim Programmstart geöffnet werden. Mit dem Öffnen werden dann verschiedene Prüfungen durchgeführt, und die Datei wird für die spezifische Verwendung zur Verfügung gestellt. Es ist hier anzumerken, dass das Öffnen einer Datei ein sehr performanceintensiver Teilschritt ist. Deshalb sollte versucht werden, gerade bei Massendatenverarbeitungen das wiederholte Öffnen (und auch Schließen) einer Datei zu vermeiden. Und genau das lässt sich mit geschicktem Einsatz von ILE-Technologien und der Kombination mit dem Schlüsselwort UsrOpn erreichen.

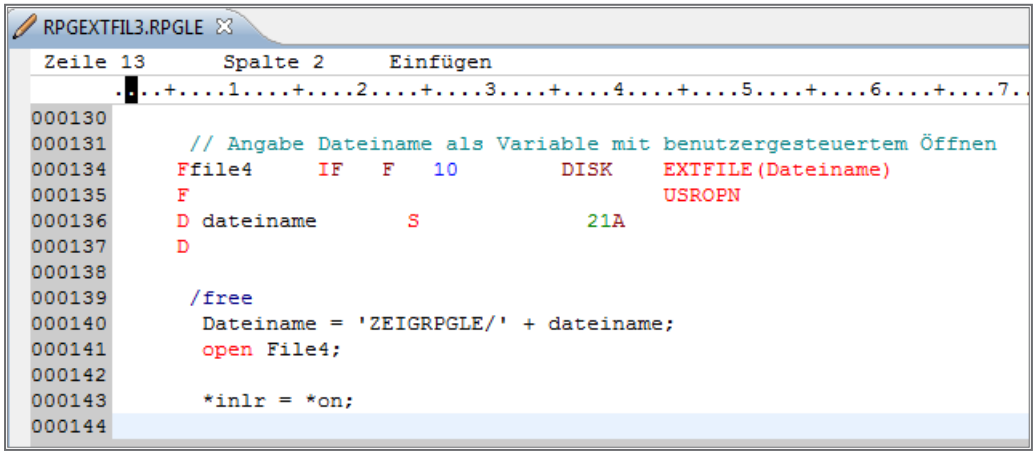
Das Schlüsselwort UsrOpn bedingt, dass der Anwendungsentwickler die so definierte Datei vor der ersten Verwendung im weiteren RPG-Ablauf mit dem Befehl OPEN öffnet. Anderenfalls führt ein Zugriff auf eine geschlossene Datei zum Programmabbruch.

Wird im Programmcode für eine UsrOpn-definierte Datei keine zugehörige OPEN-Anweisung gefunden, dann quittiert dies der Compiler mit einer Fehlermeldung. Ein Beispiel dazu zeigt nachfolgende Abbildung.



Wandlungsfehler durch fehlende OPEN-Angabe

Nachfolgendes Codebeispiel zeigt die Angabe UsrOpn zusammen mit dem zugehörigen OPEN-Befehl.



UsrOpn mit OPEN-Angabe

3.2.4
Seite 44

InfDS

Diese besondere Form der Datenstruktur beinhaltet Details zur Datei. Einsatzbereiche sind z. B. die Fehlerbehandlung bei der Arbeit mit einer Datei und das Abrufen von Informationen zu den Dateien innerhalb eines RPG-Programms. Mithilfe einer InfDS kann beispielsweise die Verarbeitung unterschiedlicher Member (Teildateien) innerhalb einer Datei gesteuert werden. Diese File Information Data Structure kann als Datenstruktur im RPG-Programm eingebunden und entsprechend verwendet werden. Der Name dieser Datenstruktur muss im RPG-Programm eindeutig sein. Es kann nach Bedarf für jede in einem Programm definierte Datei eine korrespondierende InfDS deklariert und genutzt werden.

Der Aufbau der InfDS ist immer gleich und besteht aus folgenden Informationsgruppen:

- Dateiinformationen,
- Informationen zum Öffnen,
- Ein-/Ausgabe-Informationen,
- einheitenspezifische Informationen,
- Abrufinformationen.

```
 // Beschreibung der InfDS
DKundenDS         DS
D FILE              *FILE                                  * Dateiname
D OPEN_IND                9      9N                       * Datei geöffnet?
D EOF_IND                10     10N                       * Datei EOF?
D STATUS            *STATUS                                * Statuscode
D OPCODE            *OPCODE                                * Ltzt. Opcode
D ROUTINE           *ROUTINE                               * RPG Routine
D LIST_NUM               30     37                        * Listing Zeile
D SPCL_STAT              38     42S 0                     * SPECIAL Status
D RECORD            *RECORD                                * Satzname
D MSGID                  46     52                        * Error MSGID
D SCREEN            *SIZE                                  * Screen size
D NLS_IN            *INP                                   * NLS Eingabe?
D NLS_OUT           *OUT                                   * NLS Ausgabe?
D NLS_MODE          *MODE                                  * NLS Mode?
```

Aufbau der InfDS

3.2.4
Seite 45

Über die InfDS erhalten Sie Auskunft über den Status der Datei. Beispiele: Ist die Datei geöffnet? Wurde das Dateiende erreicht?

Als Codierregeln gilt, dass auf Dateiebene als Schlüsselwort InfDS gefolgt vom Namen der Datenstruktur in Klammern (ohne Hochkommazeichen) einzugeben ist. Nachfolgender Codeausschnitt zeigt ein Beispiel für die Dateikunden.

```
FKunden    IF   E             DISK    infds(KundenDS)

 // Beschreibung der InfDS
DKundenDS         DS
D FILE              *FILE                                  * Dateiname
D OPEN_IND                9      9N                       * Datei geöffnet
D EOF_IND                10      10N                      * Datei EOF?
D STATUS            *STATUS                                * Statuscode
D OPCODE            *OPCODE                                * Ltzt. Opcode
D ROUTINE           *ROUTINE                               * RPG Routine
D LIST_NUM               30      37                       * Listing Zeile
D SPCL_STAT              38      42S 0                    * SPECIAL Status
D RECORD            *RECORD                                * Satzname
D MSGID                  46      52                       * Error MSGID
D SCREEN            *SIZE                                  * Screen size
D NLS_IN            *INP                                   * NLS Eingabe?
D NLS_OUT           *OUT                                   * NLS Ausgabe?
D NLS_MODE          *MODE                                  * NLS Mode?
```

Angabe von InfDS auf Dateiebene und Definition der Datenstruktur

Der Dateiname wird im Feld „File“ eingetragen und darf nur eine maximale Länge von acht Stellen aufweisen – ein Relikt aus alten Tagen. Mittlerweile sind nativ bis zu zehnstellige Dateinamen möglich – mit SQL auch noch mehr.

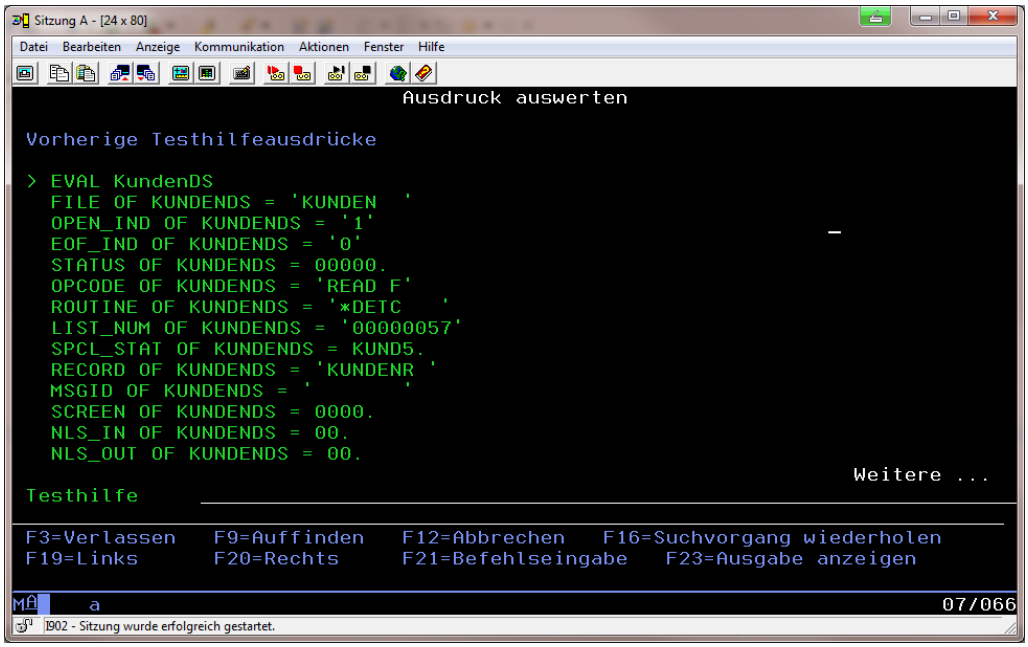
3.2.4
Seite 46

Ab Pos.	Bis Pos.	Format	Länge	Schlüssel-wort	Information
1	8	Character	8	*FILE	Die ersten acht Stellen des Dateinamens
9	9	Character	1		Anzeige Datei offen (1 = offen)
10	10	Character	1		Dateiende (1 = Dateiende)
11	15	gezont	5,0	*STATUS	Statuscode
16	21	Character	6	*OPCODE	Operationscode, der auf die Datei zugegriffen hat. Hier werden die ersten fünf Stellen dargestellt. Wie in unserem Beispiel zu sehen, wird dort READ eingetragen. Anweisungen, die länger als sechs Stellen sind, werden auf fünf Stellen gekürzt: **DELETE** → DELET **EXCEPT** → EXCPT **READPE** → REDPE **UNLOCK** → UNLCK **UPDATE** → UPDAT Die sechste Stelle liefert weitere Verarbeitungshinweise. **F** → Die letzte Anweisung wurde für die Datei selbst ausgeführt (File). **R** → Die letzte Anweisung bezog sich auf den Datensatz (Record). **I** → Bei der letzten Anweisung handelt es sich um eine implizierte Anweisung.
22	29	Character	8	*ROUTINE	Die ersten acht Stellen des Namens der Routine oder Subprozedur, in der die Anweisung ausgeführt wurde
30	37	Character	8		Wenn die OPTION(*NOSRCSTMT) angegeben wurde, dann wird hier die Source-Listung-Statement-Nummer angegeben.
38	42	gezont	5,0		Benutzerdefinierter Fehlergrund
38	45	Character	8	*RECORD	Die ersten acht Stellen des verarbeiteten Satzformatnamens

Ab Pos.	Bis Pos.	Format	Länge	Schlüssel-wort	Information
46	52	Character	7		Maschinen- oder System-nachrichtennummer
53	66	Character	14		Nicht verwendet
77	78	Binary	2		Source-Id. der Anweisungsnummer aus den Positionen 30 bis 37

Die InfDS-Angabe auf der Dateibestimmungsebene ist der eine Baustein, der andere ist die Definition der Datenstruktur selbst. Diese erfolgt in den Definitionsbestimmungen, wie die Abbildung „Angabe von InfDS auf Dateiebene und Definition der Datenstruktur“ zeigt.

Die InfDS wird vom RPG-Programm automatisch bedient, d. h., es muss sich nicht mehr um das Füllen der Datenstruktur selbst gekümmert werden. Wird z. B. lediglich ein Datensatz aus der Datei KUNDEN eingelesen und exemplarisch ein Unterbrechungspunkt gesetzt, dann kann nach dem Einlesen des Datensatzes bereits das Ergebnis in der InfDS-Datenstruktur gesehen werden.



InfDS in einer Debug-Anzeige

3.2.4
Seite 48

InfSR

Das Schlüsselwort InfSR legt fest, dass in einem Programm die Fehlerbehandlung auf Dateiebene in einer eigenen Subroutine ausgeführt wird. Fehlerbehandlungen sind ein wichtiger Bestandteil im Programmdesign und damit auch wichtig bei der Entwicklung mit RPG. Auch wenn mit RPG heute Alternativen in der Fehlerbehandlung zur Verfügung stehen, so sind dennoch gerade in älteren Programmcodes immer wieder die InfSR zu finden. Das Schlüsselwort InfSR wird auf der Ebene jeder Datei angegeben und referenziert dann auf die dort hinterlegte Subroutine zur Fehlerbehandlung. Diese Subroutine muss im späteren Verlauf des RPG-Programmcodes definiert oder eingebunden sein.

Commit

Die Commit-Steuerung der Dateien kann auf verschiedenen Bereichen eingestellt werden. Mit dem Schlüsselort Commit lässt sich in einem RPG-Programm festlegen, dass eine Datei innerhalb eines Programms mit der Commit-Steuerung genutzt werden soll. Auf diese Weise lassen sich Änderungen am Dateiinhalt in einem Journal aufzeichnen und bei Bedarf mit entsprechenden Techniken auf einen vorherigen Stand zurücksetzen. Das Schlüsselwort Commit wird ohne zusätzlichen Parameter angegeben. Ein Beispiel für die Commit-Angabe und die RolBk-Verwendung zeigt nachfolgender Code.

*RPGCOMMIT.RPGLE

```
Zeile 17      Spalte 1     Ersetzen  2 Änderungen
      .   ...+....1....+....2....+....3....+....4....+....5...
000123
000124     FKunden    uF   E             DISK    commit
000125
000126     D Status          S               1A   inz
000149
000152      /free
000153        read(e) Kunden;
000154        update KundenR;
000155
000156        if Status = '0';
000157          rolbk;
000158         else;
000159         commit;
000160        endif;
```

Commit-Angabe

3.2.4.4. Definitionsbestimmungen

In diesem Abschnitt werden die Definitionsbestimmungen vorgestellt, die in den RPG-Codes als „D"-Bestimmungen (für Definition) codiert werden.

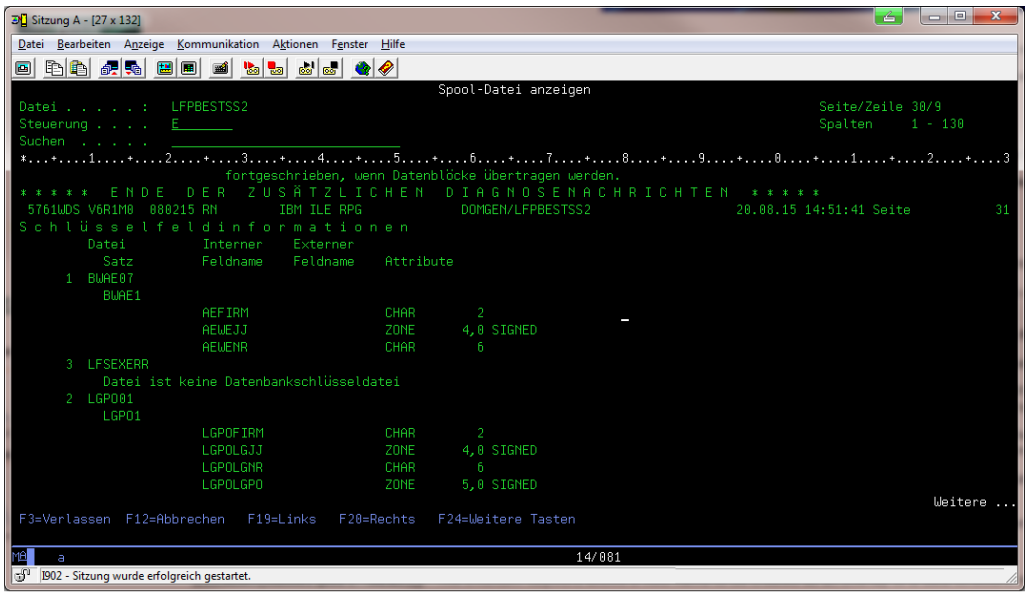
Doch welchen Sinn haben sie?

Zunächst muss ein RPG-Entwickler wissen, dass alles, was er in einem RPG-Programm machen möchte, den RPG-Grundlagen unterliegt. Damit verbunden ist das Erfordernis, dass das Programm bzw. der Compiler alle Elemente kennen und zuordnen können muss. Denn bekanntlich ist ein Programm selbst nicht wirklich mit Intelligenz versehen …

Der Anwendungsentwickler muss dafür Sorge tragen, dass alle notwendigen Angaben im Programm getätigt werden. Sämtliche Elemente, Felder, Variablen, Konstanten etc. müssen dem Programm bekannt sein. Die Definitionen dafür können extern vorgenommen werden oder müssen im Programm selbst intern beschrieben werden.

Hierzu einige Beispiele: Wenn Daten verarbeitet werden, liegen dem meist eine oder mehrere Dateien zugrunde. In den Dateien selbst sind die Felder beschrieben. Eine solche Beschreibung besteht z. B. aus dem Feldnamen und der Feldart, sollte es sich um ein alphanumerisches, numerisches oder etwa Datumsfeld handeln. Sowohl diese Angaben als auch die Angabe der Feldlänge sind wichtig. Die in den Dateien definierten Felder werden zum Zeitpunkt des Kompilierens geprüft und dem RPG zur Verfügung gestellt. Auf diese Weise sind die Dateiinhalte für das RPG-Programm definiert.

Ein Beispiel dafür findet sich in der Wandlungsliste, die in der nachfolgenden Abbildung gezeigt wird.



Felddefinitionen in der Wandlungsliste

In der Wandlungsliste sind exemplarisch die über eine Datei definierten Felder zu sehen, beispielsweise die Eintragung AEFIRM als zweistelliges alphanumerisches Feld und die Eintragung AEUEJJ als Dezimalfeld mit 4,0-Definition. Da die Datei im RPG-Programm im Zusammenhang mit den Dateibestimmungen definiert wurde, steht automatisch im RPG der in der Datei definierte Inhalt – z. B. die Felder – zur Verfügung. In einem solchen Fall darf keine Definitionsbestimmung im RPG erfolgen.

Neben diesen extern beschriebenen Elementen gibt es die RPG-Programmelemente, die nicht aus einer Datei kommen, beispielsweise Hilfsfelder innerhalb des RPG-Programms. Diese sind in keiner Datei beschrieben. Damit sie aber im RPG-Programm genutzt werden können, müssen die Hilfsfelder im RPG definiert werden. Ein Beispiel für ein solches Hilfsfeld kann ein Summenfeld sein, das im RPG berechnet wird.

Bis zum Erscheinen der vollständigen Free-Format-RPG-Variante sind alle D-Bestimmungen nach den F-Bestimmungen und vor den C-Bestimmungen zu definieren.

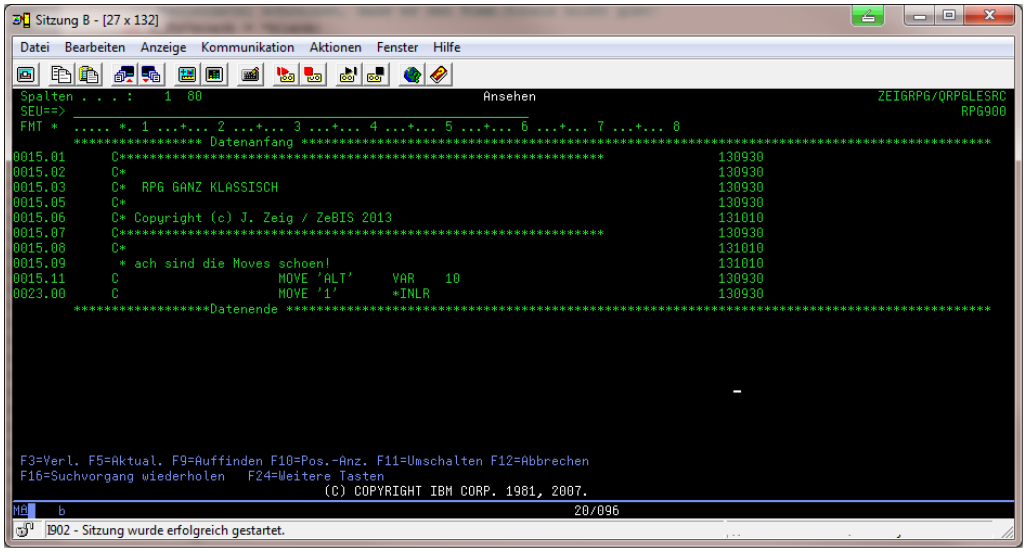
In den Definitionsbestimmungen müssen alle möglichen zu definierenden Elemente im RPG-Programm oder RPG-Modul hinterlegt werden. Dabei sind in RPG die folgenden Definitionsarten zu unterscheiden:

- Felder (Kennzeichnung „S“)
- Benannte Konstanten (Kennzeichnung „C“)
- Datenstrukturen (Kennzeichnung „DS“)
  - Programmbeschriebene Datenstrukturen
  - Extern beschriebene Datenstrukturen
  - Programmstatusdatenstrukturen
  - Dateiinformationsdatenstrukturen
  - Indikatorzuweisungen (benannte Bezugszahlen)
- Prozedur-Interface (Kennzeichnung „PI“)
- Prototypen (Kennzeichnung „PR“)

Wie es der Name bereits verrät, sollten die Definitionsbestimmungen zur Definition aller RPG-Elemente dienen, die nicht etwa automatisch, z. B. durch eine externe Definition einer Datei, zur Verfügung stehen.

In alten RPG-Programmen ist häufig ein Wildwuchs an Definitionen zu finden – sei es in den Eingabebestimmungen, in den Rechenanweisungen oder schlicht als Felddefinition in verschiedenen Rechenbestimmungen.

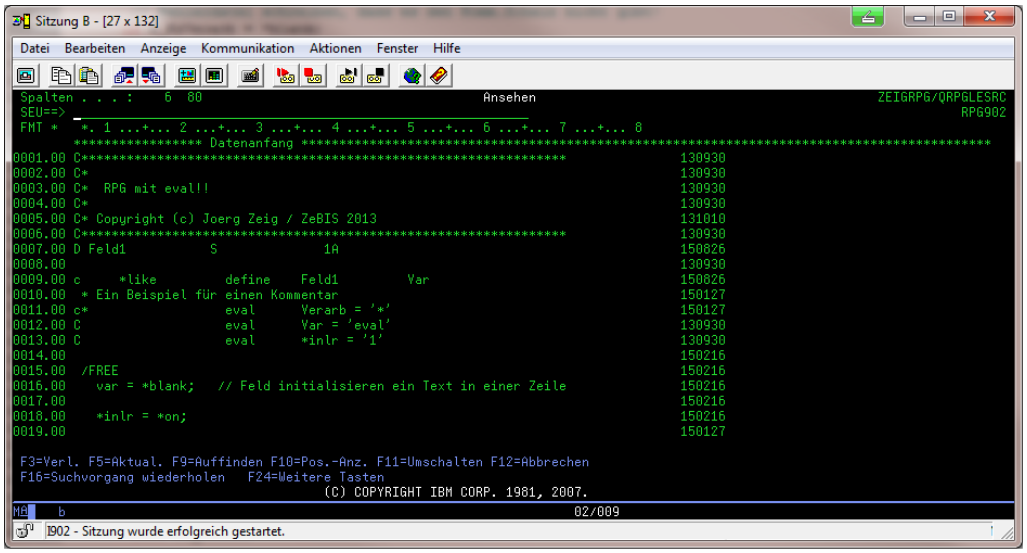
Für RPG-Neulinge wird es sicherlich hilfreich sein zu wissen, wie früher Definitionen in RPG durchgeführt wurden. Anhand der nachfolgenden Beispiele ist ersichtlich, wie es heute ganz sicher nicht mehr gemacht werden sollte!



Klassische Form der Definition auf Rechenbestimmungsebene

In oben stehender Abbildung erfolgte die Felddefinition für die Variable „VAR“ direkt in der Rechenbestimmung. Mit der Angabe „10“ wird dem RPG-Programm kenntlich gemacht, dass es sich um ein zehnstelliges Alphafeld handelt.

In der nächsten Abbildung ist ein Beispiel für den Einsatz von *LIKE-Anweisungen aufgeführt. Hier wird mit der Anweisung *LIKE DEFINE festgelegt, dass die Variable „VAR1“ die Definition nutzen soll, die zuvor auch für die Variable „FELD1“ festgelegt wurde.



Verwendung der D-Bestimmung

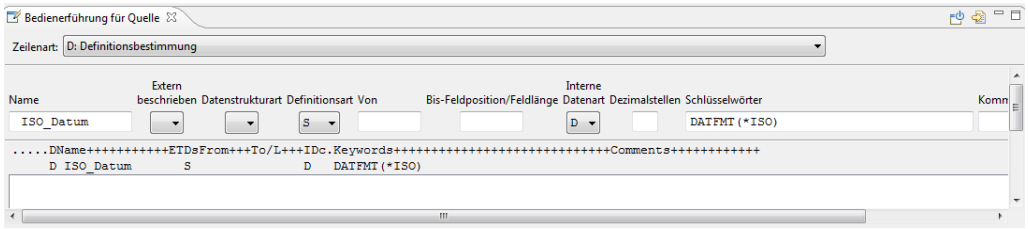
3.2.4
Seite 52

IBM hat die *LIKE DEFINE-Anweisungen in der Free-Format-Version zurückgezogen. Spätestens mit dem Free-Format sollte sich jeder Anwendungsentwickler die einheitliche Definition der RPG-Elemente aneignen.

Auch die Angabe von Feldgruppen mutet in den alten RPG-Varianten eigenartig an. Diese mussten in den E-Bestimmungen hinterlegt werden. Wie es sich anhand dieser wenigen Beispiele bereits erahnen lässt – in älteren RPG-Versionen war es dem Anwendungsentwickler in weiten Teilen selbst überlassen, wo und wie er Definitionen vorgenommen hat. Sowohl eine einheitliche Vorgehensart als auch ein direktes und schnelles Auffinden der Definitionsangaben lagen nicht vor.

Nach dieser kleinen Exkursion in die Vergangenheit des RPG im Zusammenhang mit Definition soll im Weiteren der modernere Ansatz betrachtet werden. IBM hat die Vereinheitlichung der Definitionsbestimmungen mit RPG IV eingeführt. Diese sollen die vielen möglichen Stellen der Definition von RPG-Elementen vereinheitlichen. Ein definierter Bereich für möglichst alle Definitionsangaben sollte das Ziel sein.

Die Definitionsbestimmungen beginnen sowohl im alten Free-Format-RPG als auch in der Fix-Format-Variante mit dem „D“ in Spalte 6. Es lassen sich dort alle notwendigen Deklarationen definieren. Natürlich bietet IBM auch Unterstützung für das Codieren mit der Bedienerführung an, die sich wie gewohnt mit der F4-Taste aufrufen lässt. Die Bedienerführung steht sowohl in der SEU-Variante als auch im LPEX-Editor zur Verfügung.



Bedienerführung auf Ebene der D-Bestimmung

Die unterschiedlichen Elemente sind durch die Definitionsart und Schlüsselwörter dediziert definierbar.

Aufbau der Definitionsbestimmung im Fix-Format:

- Spalte 6:
  - „D“ für Definitionsbestimmung
- Spalten 7 – 21: Name des RPG-Elements
  - Hier steht der Name des zu definierenden Elements bzw. der Variablen. Dieser sollte eindeutig gewählt werden und kann bei globalen und lokalen Variablen durchaus doppelt angegeben werden. Besonderheit ist, dass der Feldname hier nicht zwingend linksbündig, also beginnend in Spalte 7, beginnen muss. In Abhängigkeit des zu definierenden Elements bzw. der Variablen kann diese Spalte leer bleiben oder muss mit einem Wert hinterlegt werden. Details finden sich im späteren Verlauf, wenn die Bedeutung und die Spezifika der unterschiedlichen Elemente und Variablen behandelt werden.
- Spalte 22: Externe Beschreibung
  - Hier wird mit dem Wert „E“ auf eine externe Beschreibung verwiesen. Diese wird beispielsweise bei externen Datenstrukturen genutzt oder kann auch für Feldreferenzen zum Einsatz kommen. Gültige Werte sind „Blank“ und „E“.
- Spalte 23: Art der Datenstruktur
  - In dieser Spalte wird die Art der Datenstruktur angegeben.
  - Sie kommt nur in Verbindung mit speziellen Elementen (Datenstrukturen) zum Einsatz. Hier lässt es sich festlegen, ob es sich um eine Programmstatus- (S) oder eine Datenbereichsdatenstruktur (U) handelt.
- Spalten 24 – 25: Elementart
  - In dieser Spalte wird angegeben, ob es sich um ein Feld („S“ = Standalone-Feld), eine Konstante („C“), einen Prototypen („PR“) oder ein Prozedur-Interface („PI“) handelt. Wenn Sie Unterfelder zu einer Datenstruktur definieren, dann muss dieses Feld zwingend leer bleiben!
- Spalten 26 – 32: Von-Position
  - Hier wird die „Von-Position“ angegeben. Allerdings beschränkt sich die Eingabe auf Unterfelder von Datenstrukturen. In dem Fall wird die absolute Startposition des Feldes angegeben. Bei allen anderen Definitionen kann diese Spalte leer bleiben.

3.2.4
Seite 54

- Spalten 33 – 39: Bis-Position
  - Die Angabe „Bis-Position“ bzw. „Länge“ legt die Länge des Elements fest. Soll z. B. ein Feld definiert werden, dann wird hier die absolute Länge des Feldes hinterlegt, wobei zwingend die Beschränkung hinsichtlich der maximalen Länge der RPG-Elemente berücksichtigt werden muss. Diese kann von Release zu Release variieren.
- Spalte 40: Datentyp
  - In dieser Spalte wird der Datentyp festgelegt. Folgende Eingaben sind dort möglich:
    A = Alphanumerisch
    B = Binär
    C = UCS-2 (2 Bytes Unicode)
    D = Datumsfeld
    F = Gleitkomma
    G = Grafisches Format
    I = Integer
    N = Indikator
    O = Objekt (eingesetzt i. d. R. für JAVA-Methoden)
    P = Gepacktes Dezimalformat
    S = Gezontes Dezimalformat
    T = Zeitfeld
    U = Unsigned (positive Integerwerte)
    Z = Zeitmarke
    * = Pointer
- Spalten 41 – 42: Anzahl der Dezimalstellen
  - Für numerische Werte sind hier die Dezimalstellen einzugeben.
- Spalte 43: Reserviert
  - Diese Spalte hat derzeit keine Bedeutung.
- Spalten 44 – 80: Schlüsselwörter
  - Mithilfe der unterschiedlichen Schlüsselwörter lassen sich verschiedentliche Verhaltensweisen und Eigenschaften definieren. Details dazu sind später zu finden.
- Ab Spalte 81: Kommentare
  - Hier können Kommentare zur Zeile eingegeben werden.

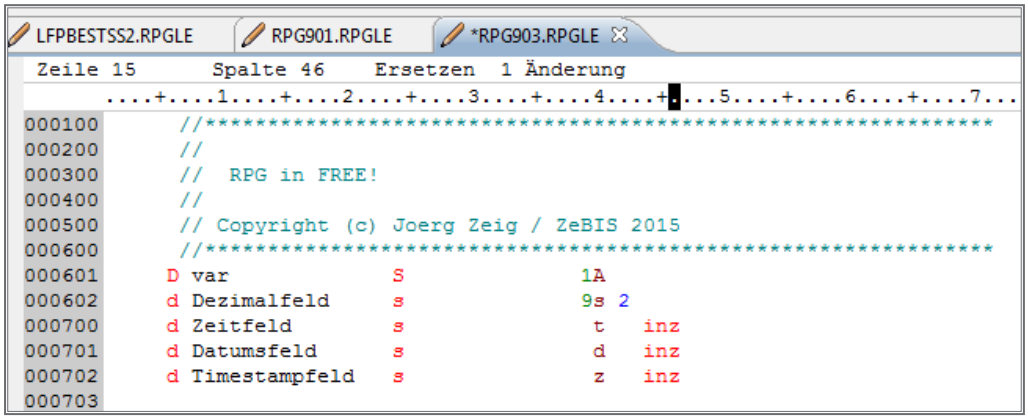
So viel zum Aufbau der D-Karten und der einzugebenden Informationen.

In den früheren RPG-Varianten wurden gelegentlich Referenzen verwendet, die im Fix-Format über die Eingabe *LIKE DEFINE definiert wurden. Dies diente dazu, auf ein bestehendes Feld zu referenzieren und dessen Beschreibung für die eigene Variable innerhalb eines RPGs zu nutzen. Die *LIKE DEFINE-Anweisung ist im Free-Format verschwunden, die Definitionsmöglichkeit der Referenzierung ist jedoch nach wie vor existent – nur in abgewandelter Form. Diese wird in den D-Bestimmungen z. B. als Standalone-Feld mit der Referenz auf ein bestehendes Feld mittels des Schlüsselwortes „LIKE“ hinterlegt. Nachfolgende Abbildung zeigt dazu ein Beispiel.

Im Folgenden sind Beispiele für Definitionsbestimmungen aufgeführt.

Standalone-Feld/-Variable
Bei einem zu definierenden Feld bzw. einer Variablen müssen folgende Angaben getätigt werden:

- Name des Feldes / der Variablen,
- Definitionsart „S“ (Standalone-Feld),
- Bis-Feldposition (Länge des Feldes),
- interne Feldart,
- Dezimalstellen (bei Zahlenwerten).



Beispiele für Feld-/Variablendefinitionen

Im vorhergehenden Code-Beispiel sind einige unterschiedliche Variablen definiert. Bei der ersten Variablen „var“ handelt es sich um ein einstelliges alphanumerisches Feld.

Das Feld „Dezimalstellen“ ist mit neun Stellen und zwei Nachkommastellen definiert.

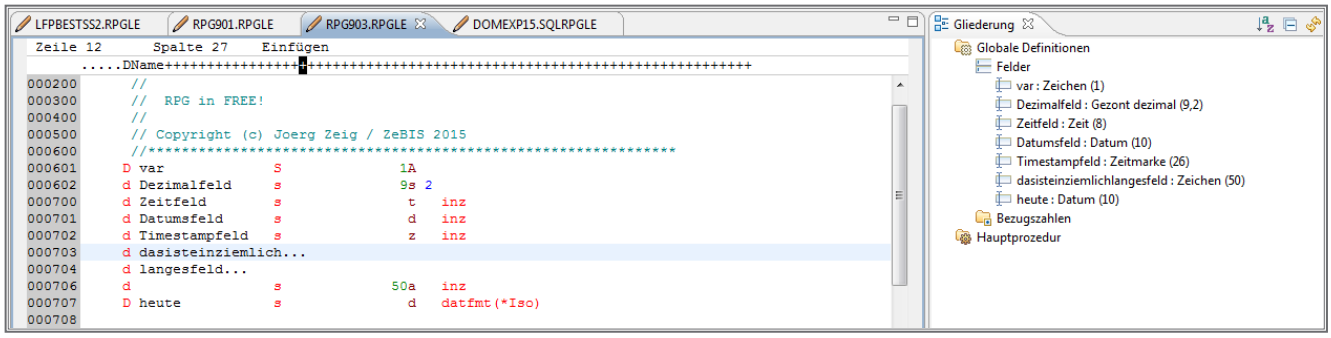
3.2.4
Seite 56

Das „Zeitfeld“ ist als Zeitfeld (Angabe t) festgelegt. Mittels des Schlüsselwortes „inz“ wird definiert, dass die Variable initialisiert werden soll. Es ist ersichtlich, dass die Angabe der Spalten für die unterschiedlichen Variablenarten / Standalone-Felder ähnlich ist und sich im Wesentlichen durch die Art sowie die Angabe der Dezimalstellen und optional der Schlüsselwörter unterscheidet.

Ein Hinweis zum Schlüsselwort „inz“: Die Initialisierung kann beim Erstaufruf auch für alphanumerische und numerische Felder hinterlegt werden – im Code-Beispiel sollten lediglich die unterschiedlichen Eingabemöglichkeiten demonstriert werden.

Eine Besonderheit ist im Hinblick auf die Länge des Elementnamens (hier des Feldnamens) zu erwähnen: IBM hat aufgrund der Vergangenheit einige Verunsicherung in Bezug auf die maximale Länge eines Feldnamens (nicht des Werts eines Feldes) ausgelöst. Während in alten RPG-Versionen ein Feldname innerhalb von RPG auf zehn bzw. 15 Stellen beschränkt war, kann heute mit Feldnamen gearbeitet werden, die bis zu 4.096 Stellen lang sein können. Natürlich ist ein solch langer Feldname außergewöhnlich, doch z. B. bei der Beschäftigung mit der Verarbeitung von externen Daten ist dies immer wieder dienlich. Im folgenden Code-Ausschnitt ist ein Beispiel für die Definition eines langen Feldnamens gezeigt. Zu beachten ist dabei Folgendes: Die Zeilenumbrüche und Ende-Angaben eines Feldnamens müssen mit drei aufeinanderfolgenden Punkten codiert werden. Die Pflichtspalten sind dabei nur in der ersten Zeile einzugeben.

In nachfolgender Abbildung ist ein Feld „dasisteinziemlichlangesfeld“ definiert. Der Feldname selbst ist deutlich länger als der Eingabebereich für den Namen in einer Zeile. Bitte beachten Sie die Gliederungssicht auf der rechten Seite – dort findet sich der Feldname, wie er im RPG intern genutzt wird.



Beispiel für einen langen Feldnamen

Datenstruktur

Bei einer Datenstruktur handelt es sich um eine Option, mit der eine Fülle an Anforderungen umgesetzt werden kann. Genauere Informationen zu den Besonderheiten der Datenstrukturen finden sich in einem anderen Kapitel. Hier soll es exemplarisch um die Definition der Datenstruktur in den Definitionsbestimmungen gehen.

Die Mindestangaben für eine Datenstruktur bestehen aus zwei Zeilen:

1. der Kopfzeile – diese beinhaltet den Namen der Datenstruktur;
2. den Unterfeldangaben (Mehrfachvorkommen) – hier werden die Details zu den Unterfeldern einer Datenstruktur aufgenommen.

Datenstrukturen können zudem für die unterschiedlichsten Bereiche genutzt werden. Im nachfolgenden Beispiel wurde eine Systemdatenstruktur definiert, die den Namen des Systems trägt und die Felder für das Programm, die Fehlernachricht, die Work-Station und den Benutzer beinhaltet.

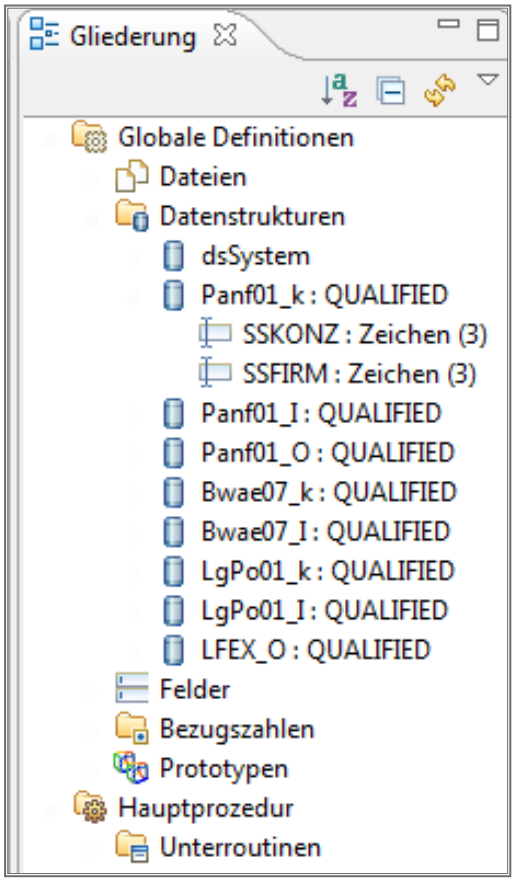
```
006900      // System-Datenstruktur
007000
007100     d dsSystem       sds
007200     d  aThisProgramm          1     10
007300     d  aerrMsgID             40     46
007400     d  aThisWorkStn         244    253
007500     d  aThisUser            254    263
007600
007700
007800     d Panf01_k       E DS                  ExtName(PBestSS2:*Key)
007900     d                                      Qualified
008000     d Panf01_I       E DS                  ExtName(PBestSS2:*input)
008100     d                                      Qualified
008200     d Panf01_O       E DS                  ExtName(PBestSS2:*output)
008300     d                                      Qualified
```

Beispiele für allgemeine Datenstrukturen

Weiterhin sind im Code-Ausschnitt Beispiele für die Definition von Datenstrukturen zu sehen, die für das Arbeiten mit Dateien bestimmt sind. So wird die Datei PbestSS2 mittels einer Schlüsseldatenstruktur verarbeitet. Dabei wird mit dem Schlüsselwort ExtName auf die extern beschriebene Datei PBestSS2 referenziert. Mit dem Sonderwert „*Key“ wird festgelegt, dass die Datenstruktur PANF01_K als Schlüsseldatenstruktur für den Zugriff auf die angegebene Datei zu verwenden ist.

3.2.4
Seite 58

Basierend auf den Angaben in der Gliederung werden die Schlüsselfelder der Datei aufgelistet.



Darstellung der Datenstruktur in der Gliederung

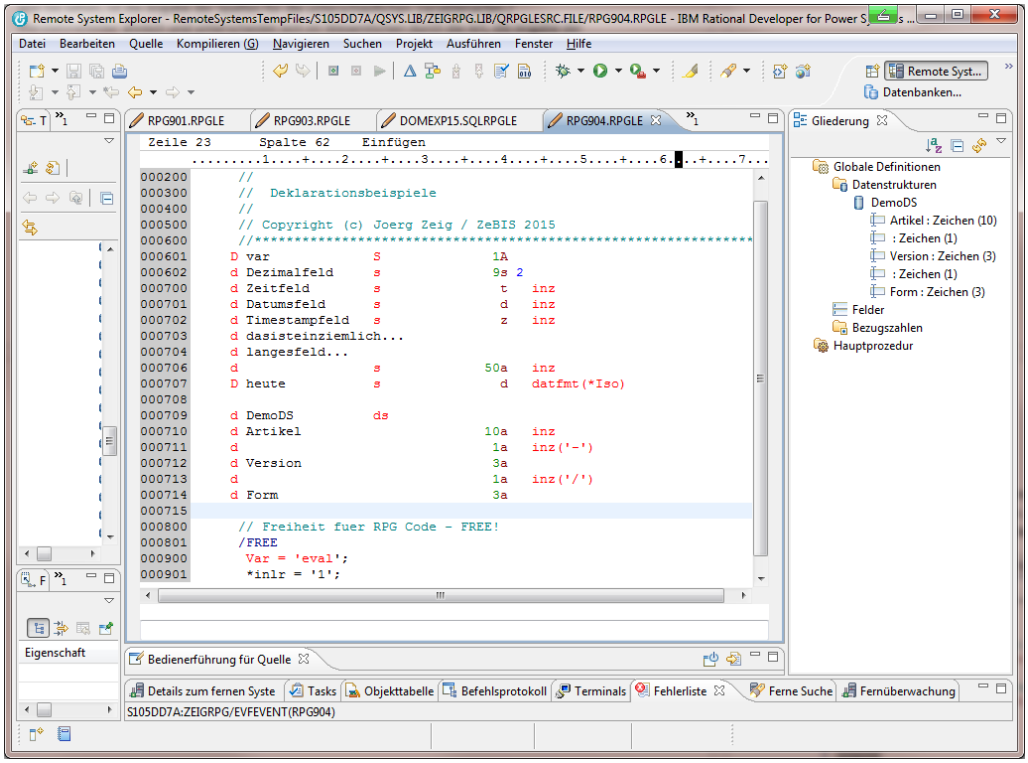
Die Datenstruktur PANF01_K beinhaltet die beiden Unterfelder SSKONZ und SSFIRM. Bei diesen handelt es sich um die Schlüsselfelder der Datei.

Hinweis zum vorherigen Beispiel: Es ist die Erweiterung „Qualified“ zu beachten, die in allen Datenstruktureinträgen genutzt wird. Mit diesem Schlüsselwort wird festgelegt, dass der Inhalt der Datenstruktur (die Felder) qualifiziert anzugeben ist. Damit wird beispielsweise das Schlüsselfeld SSKONZ aus der Datei PBESTSS2 im RPG-Programm mit dem qualifizierten Namen codiert. Dieser besteht aus dem Namen der Datenstruktur, gefolgt vom Trennzeichen in Form eines Punktes, gefolgt vom eigentlichen Feldnamen. In der Praxis sieht das z. B. so aus: PANF01_K.SSKONZ =’001’;

Vorteil des qualifizierten Feldnamens: Sie haben in der Quelle immer direkt eine Zuordnung, ob Sie ein Feld zum Zugriff auf eine Datei (als Schlüsselfeld) nutzen, ob Sie es zum Einlesen verwenden (*INPUT) oder ob Sie es zum Fortschreiben einer Datei einsetzen (*OUTPUT). Dies ist keine IBM-Vorgabe, sondern lediglich eine Form des Codierens im RPG.

An dieser Stelle nur dies eine Beispiel für die Definition einer Datenstruktur. Die verschiedenen Ausprägungen der Datenstrukturen werden später noch genauer betrachtet.

Wenn eine Datenstruktur definiert werden soll, deren Unterfelder aus Konstanten bestehen, dann braucht dafür kein Name eingetragen zu werden. Nachfolgende Abbildung zeigt ein Beispiel hierzu.



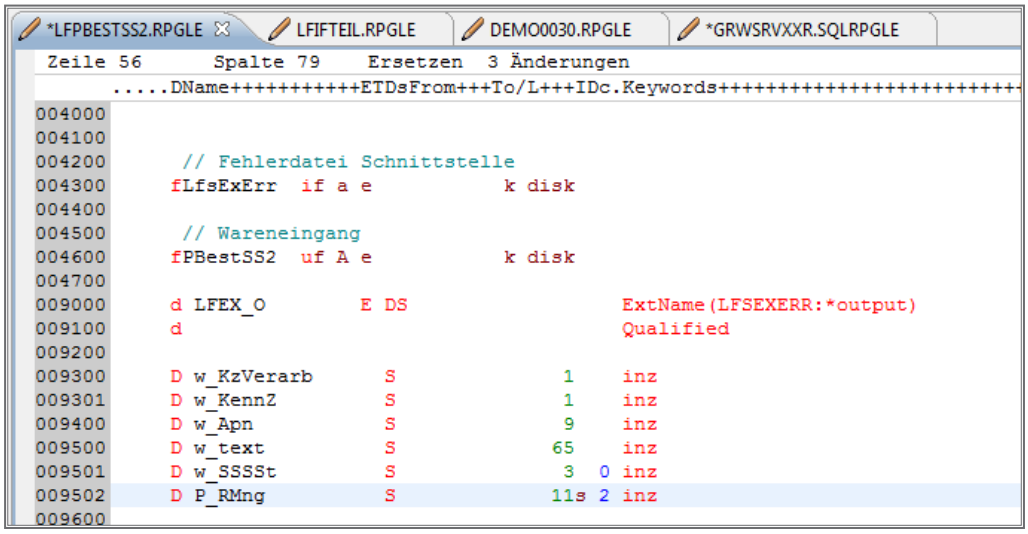
Beispiel für eine Datenstruktur mit konstanten Unterfeldern

Hier wird die Datenstruktur „DemoDS“ mit den variablen Unterfeldern „Artikel“, „Version“ und „Farbe“ definiert. Diese sind durch Trennkonstanten separiert eingegeben, deren Werte über Definitionen von Konstanten und ohne Eingabe des Namensbereichs definiert wurden. Auch hier ist die Anzeige in der Gliederung zu beachten.

3.2.4
Seite 60

Globale und lokale Variablen

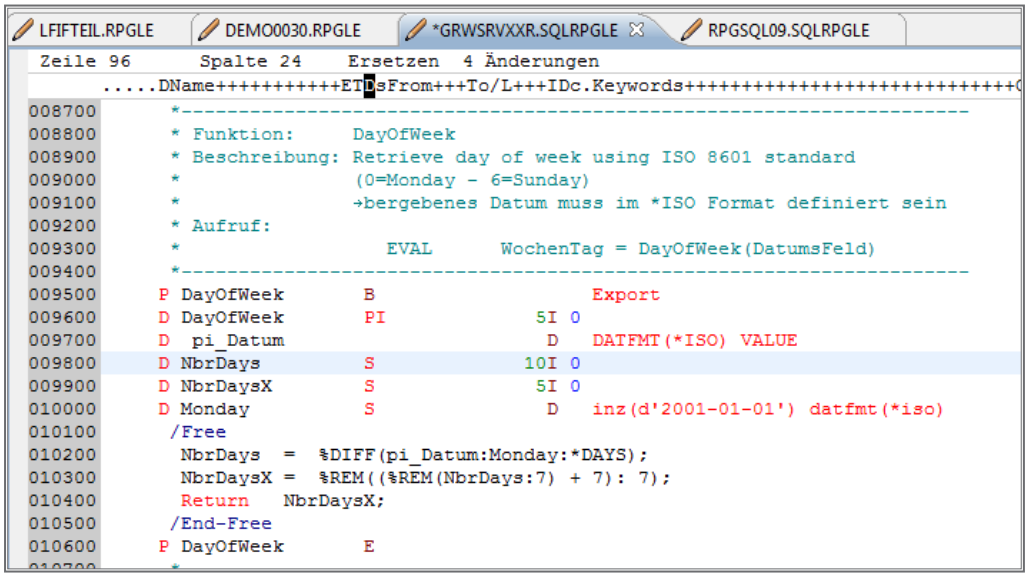
Die Definitionsbestimmungen können auf globaler Ebene oder bei Bedarf auf lokaler Ebene (Prozedurebene) durchgeführt werden. Die globalen Definitionen folgen auf die H- oder F-Bestimmungen. Folgende Abbildung zeigt ein Beispiel für eine globale Definition.



Globale Definition

Die auf globaler Ebene definierten Elemente stehen im gesamten RPG zur Verfügung. Wenn Prozeduren innerhalb eines RPG-Codes verwendet werden, dann können in den Prozeduren ebenfalls D-Bestimmungen definiert werden. Dabei handelt es sich um sogenannte „lokale“ Variablen, die lediglich für die Nutzung innerhalb der Prozedur vorgesehen sind.

Auch hierzu ein einfaches Beispiel. Zu beachten ist: Die Angaben unterscheiden sich lediglich durch die Platzierung in der Prozedur von der globalen Definition.



Lokale Definition

Bitte beachten Sie die vor den Definitionsbestimmungen gesetzte P-Anweisung. Diese steht für eine Prozedur, die später noch genauer betrachtet werden soll.

3.2.4.4.1 Schlüsselwörter für Definitionsbestimmungen

Über Schlüsselwörter wurde bereits bei den Dateibestimmungen berichtet. Auch für die Definitionsbestimmungen bietet IBM unterschiedliche Steuermöglichkeiten auf Basis der Schlüsselwörter an. Nachfolgend einige der häufig genutzten:

inz
Mit dem Schlüsselwort „inz“ wird das RPG-Programm dazu veranlasst, den Wert der definierten Variablen mit dem angegebenen Initialisierungswert zu belegen.

```
009300     D w_KzVerarb      S              1    inz
009301     D w_KennZ         S              1    inz(' ')
009400     D w_Apn           S              9    inz
009500     D w_text          S             65    inz
009501     D w_SSSSt         S              3  0 inz(0)
009502     D P_RMng          S             11s 2 inz
```

80 Beispiel für die Verwendung von „inz“

„inz“ kann auf den unterschiedlichen RPG-Element-Ebenen angegeben werden. So lässt sich z. B. der Inhalt einer Datenstruktur genauso mit dem „inz“-Befehl initialisieren wie ein definiertes Feld bzw. eine Variable. Die zuletzt genannte Form ist in der vorherigen Abbildung gezeigt. Hier kann dem System mitgeteilt werden, dass es anhand der Feldbeschreibung selbst die Basisinitialisierung durchführen soll. Das bedeutet, dass die als „Charakter“ definierte Variable w_Kzverarb aus dem vorherigen Beispiel mit dem Wert „Blank“, dem Basisinitialisierungswert für alphanumerische Felder, initialisiert wird.

Das Feld p_RMng (dezimal, 11,2-stellig) würde ohne weitere Angabe mit dem Initialisierungswert 0 vorbelegt.

Wenn Sie selbst den Initialisierungswert mitgeben wollen, dann ist dies unmittelbar nach dem Schlüsselwort „inz“ in runden Klammern möglich. Achten Sie dabei allerdings auf die korrekte Schreibweise! Alphanumerische Initialisierungswerte müssen in einfachen Hochkomma stehen. Numerische Initialisierungswerte müssen hingegen ohne Hochkomma in den beiden runden Klammern eingegeben werden.

Varying

IBM hat mittlerweile die Option geschaffen, Felder mit variabler Länge innerhalb von RPG verarbeiten zu können. Mit dem Erscheinen von BIFs, wie z. B. %Trim und anderen RPG-Erweiterungen, lassen sich RPG-Felder mit variabler Länge ideal nutzen.

RPG-Elemente mit einer variablen Länge werden mit dem Schlüsselwort „Varying“ definiert. Wenn z. B. ein Varying-Feld definiert wird, dann besteht es aus zwei Bereichen:

1. der aktuellen Länge, die über einen 2-Byte-Integerwert definiert ist, und
2. dem aktuellen Wert.

Die Verwendung von Varying ist in nachfolgender Abbildung gezeigt. Die Definition eines solchen Feldes erfolgt auf dieselbe Art wie ein „normales“ Feld mit einer fixen Länge. Lediglich die Verwendung des Schlüsselwortes „Varying“ ist hier entscheidend.

LFPBESTSS2.RPGLE | LFIFTEIL.RPGLE | DEMO0030.RPGLE | *GRWSRVXXR.SQL

Zeile 26 Spalte 54 Ersetzen 2 Änderungen

```
.....DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++
001800      //---------------------------------------------------
001900      //    S T E U E R - B E S T I M M U N G E N
002000      //---------------------------------------------------
002100     h datfmt(*EUR)
002200     h datedit(*DMY.)
002300     h timfmt(*HMS:)
002400     h decedit('0,')
002500     h langid('DE')
002600     h option(*SRCSTMT:*NoDebugIO)
002700     h dftactgrp(*NO) actgrp(*CALLER)
002800     h copyright('Joerg Zeig 10/2014')
002801
002802     D Anzahl          S              11P 2
002803     D IndAnzahl       S               5I 0
002804     D String          S             256A   Varying
002805     D                                      inz
002806     D ParTable        S              10A   inz('ARTIKEL')
002807     D ParArtikelNr    S              22A   inz('5000')
002808     D ParMandant      S              10A   inz('001')
002809
```

Varying-Angabe für ein Feld

Bitte beachten Sie in der Definition auch die zweite Zeile, die für die Definition der Variablen „String“ genutzt wird. Das Initialisieren bietet sich in jedem Fall an!

3.2.4
Seite 64

Bei der Angabe der Initialisierung in einer zweiten Zeile handelt es sich um eine Codierform von mir – Sie jedoch können nach Belieben die Initialisierungsangabe auch als zweites Schlüsselwort nach der Angabe „Varying“ codieren.

```
p DayName          B                   Export
d DayName          pi            32a   Varying
d pi_InputDate                     d   Const DatFmt(*iso)

 * Nur deutsche Version...
d                  ds
d Days                           70a   Inz('Montag    +
d                                         Dienstag   +
d                                         Mittwoch   +
d                                         Donnerstag+
d                                         Freitag    +
d                                         Samstag    +
d                                         Sonntag    ')
d Day                            10a   Dim(7) Overlay(Days)
```

Beispiel für Varying

ExtName

Die Möglichkeiten der Definition beziehen sich in RPG u. a. auch auf das Referenzieren auf extern beschriebene Daten. In diesem Zusammenhang kann auf der Ebene der Datenstrukturen das Schlüsselwort „ExtName“ verwendet werden.

```
006900      // System-Datenstruktur
007000
007100     d dsSystem       sds
007200     d  aThisProgramm          1     10
007300     d  aerrMsgID             40     46
007400     d  aThisWorkStn         244    253
007500     d  aThisUser            254    263
007600
007700
007800     d Panf01_k      E DS                  ExtName(PBestSS2:*Key)
007900     d                                     Qualified
008000     d Panf01_I      E DS                  ExtName(PBestSS2:*input)
008100     d                                     Qualified
008200     d Panf01_O      E DS                  ExtName(PBestSS2:*output)
008300     d                                     Qualified
```

ExtName-Beispiel

Das Schlüsselwort „ExtName“ kann sowohl in Verbindung mit dem einfachen als auch in Verbindung mit dem qualifizierten Dateinamen (in Form der Bibliothek/Datei) codiert werden. Im Fall der qualifizierten Angabe sind Bibliothek und Datei durch einen Schrägstrich (/) voneinander zu trennen.

Neben dem Dateinamen lässt sich die Form der Verwendung der Datenstruktur definieren. Hier sind folgende Werte gültig:

- *ALL
  Verwendung aller Felder
- *KEY
  Extrahieren der Schlüsselfelder basierend auf der Dateibeschreibung (automatisch)
- *INPUT
  Einbinden der eingabefähigen Felder der Datei
- *OUTPUT
  Einbinden der ausgabefähigen Felder der Datei

3.2.4
Seite 66

Qualified

Das Schlüsselwort „Qualified“ wurde bereits zuvor erwähnt. Die Angabe der qualifizierten Verwendung bezieht sich auf Datenstrukturen, deren Elemente bzw. Felder qualifiziert anzugeben sind. „Qualified“ wird auf der Ebene der Datenstruktur angegeben und kann zusammen mit anderen Schlüsselwörtern genutzt werden.

```
007700
007800     d Panf01_k      E DS                  ExtName(PBestSS2:*Key)
007900     d                                     Qualified
008000     d Panf01_I      E DS                  ExtName(PBestSS2:*input)
008100     d                                     Qualified
008200     d Panf01_O      E DS                  ExtName(PBestSS2:*output)
008300     d                                     Qualified
008400
008500     d Bwae07_k      E DS                  ExtName(Bwae07:*key)
008600     d                                     Qualified
008700     d Bwae07_I      E DS                  ExtName(Bwae07:*input)
008800     d                                     Qualified
008801
008802     d LgPo01_k      E DS                  ExtName(LgPo01:*key)
008803     d                                     Qualified
008804     d LgPo01_I      E DS                  ExtName(LgPo01:*input)
008805     d                                     Qualified
008900
009000     d LFEX_O        E DS                  ExtName(LFSEXERR:*output)
009100     d                                     Qualified
009200
```

Verwendung von Qualified

Der qualifizierte Name besteht aus dem Namen der Datenstruktur, gefolgt vom Trennzeichen in Form eines Punkts, gefolgt vom eigentlichen Feldnamen. Das sieht im RPG-Code z. B. so aus:

```
018508
018509          Bwae07_k.AeFirm = p_Firm;
018510          Bwae07_k.AeWeJJ = p_Wejj;
018511          Bwae07_k.AeWeNr = p_Wenr;
018512
```

Beispiel für die Verwendung qualifizierter Datenstrukturen

3.2.4.5 RPG-Rechenbestimmungen

Die bisherigen Bestimmungsarten in RPG dienen der Vorbereitung für das Erstellen eines Programms, das wahlweise Daten verarbeiten, Steuerungen übernehmen oder für andere Zwecke zum Einsatz kommen kann.

Die Vielzahl der RPG-Zeilen wird sich in der Regel auf Basis der sogenannten Rechenbestimmungen verteilen. Die Rechenbestimmungen (C-Bestimmungen) machen das Programm aus, und sie beinhalten neben dem Programmfluss auch die Umsetzung der Anforderungen für das Arbeiten mit Daten, System-i-Ressourcen, Bedingungsdefinitionen und vielem mehr.

Wie bereits erwähnt, sind aus der Historie heraus unterschiedlichste RPG-Rechenbestimmungen in den Unternehmen im Einsatz. Begonnen mit dem klassischen RPG II und dem RPG III, gefolgt vom erweiterten Faktor 2, dem RPG IV (ILE) bis hin zu Free-Format-RPG sind diese Varianten heute vorzufinden. Da an dieser Stelle eine Beschäftigung mit den neueren und modernen Ansätzen mit RPG erfolgen soll, sei lediglich auf die Klassik-RPG-Welt verwiesen. Im folgenden Kapitel sollen indes die RPG-Techniken und -Rechenbestimmungen behandelt werden, die heute auch auf Rat der IBM genutzt werden sollten.

Free-Format-RPG hat sich bisher noch nicht weit verbreiten können. Das ist umso mehr ein Grund dafür, sich detailliert damit zu beschäftigen sowie die Grundlage für eine effektive und moderne Arbeit mit RPG zu bilden.

Zunächst soll auf die Differenzierung der ILE-Komponenten (Module, Serviceprogramme) verzichtet und sich stattdessen auf die Kodierung von reinen RPG-Programmen beschränkt werden. Denn in diesem Kapitel geht es primär ums Kodieren in RPG, das Verständnis der Verarbeitung von Daten, den Einsatz von Dateien und den Programmfluss.

Heute bietet RPG durchaus vergleichbare Methoden und Techniken der Anwendungsentwicklung, wie sie bereits von anderen, „modernen" Programmiersprachen bekannt sind – wobei hier „modern" bewusst in Anführungszeichen gefasst ist, denn es ist letztlich Auslegungssache, ob beispielsweise Java oder C als „modern" zu bezeichnen ist. Jede Programmiersprache hat ihre Stärken und Schwächen – und die Stärke von RPG ist unbenommen der Einsatz zusammen mit der Datenbank DB2/UDB des System i. Leider aber ist die Stärke von RPG zugleich auch dessen massivste Schwäche – denn im Regelfall wird RPG für den Einsatz mit System i genutzt und nicht zusammen mit anderen Plattformen oder Datenbanken.

3.2.4
Seite 68

Wenn Sie als Anwendungsentwickler bereits über Erfahrung in C, COBOL, PL/1 oder in einer anderen Programmiersprache im Free-Format verfügen, dann sollte Ihnen das Erlernen von RPG recht leicht fallen. Aber auch diejenigen, die noch keine Kenntnisse in der Programmierung haben, werden nach einer gewissen Einarbeitungszeit die Vorzüge und die Arbeitsweise von RPG schätzen lernen.

Allen Unkenrufen zum Trotz: RPG ist eine durchaus moderne Entwicklungssprache, die für Anwendungen in den Jahren 2015 und den folgenden durchaus eingesetzt werden kann und deren Basis für die nächsten Jahre das Rückgrat für Unternehmen und Geschäftsprozesse ist.

Da noch nicht alle Unternehmen die neuesten Betriebssystemversionen des IBM i nutzen (ab 7.1 mit TR Stufe 7), soll zunächst erläutert werden, wie die Rechenbestimmungen im Free-Format-RPG gestartet werden.

Die IBM i-Versionen vor 7.1 und vor TR7 nutzen die erste Form des Free-Format-RPG, die sich ausschließlich auf das Kodieren der Rechenbestimmungen im freien Format beschränkt. Da die Vielzahl der Installationen über diese Versionen verfügt, sei an dieser Stelle die dort notwendige Startinformation für den Free-Format-Block gegeben.

Der Beginn eines Blocks bzw. einer Sektion im Free-Format erfolgt, indem zuerst die Startangabe für das Free-Format in Form von „/Free“, an Stelle 7 des Programmcodes beginnend, vorgenommen wird. Der Rest in dieser Zeile (Positionen 12 bis 80) bleibt leer.

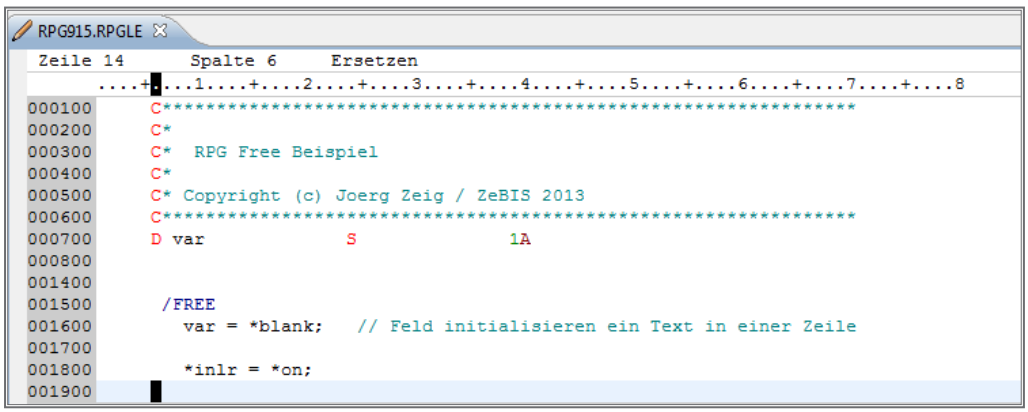
RPG im Free-Format ist nicht Case Sensitive – das bedeutet auch ein Freisein in Bezug auf die Schreibweise der Free-Format-Startanweisung wie auch aller übrigen RPG-Eingaben (Ausnahmen sind Konstanten, die jedoch erst später behandelt werden). Folglich kann als Startanweisung für das Free-Format „/FREE“, „/Free“ oder „/free“ verwendet werden.

Der Free-Format-RPG-Block wird mit der Angabe der Anweisung „/End-free“ abgeschlossen. Wenn darauf kein Fixformatfolgt, dann braucht es für den Block des freien Formats keine Ende-Angabe.

Innerhalb der /Free- und /End-free-Anweisung wird der Free-Format-RPG-Code platziert. Hier spielt sich das Leben des RPG-Programms ab. Die Angabe der einzelnen Anweisungen kann ab Stelle 8 beliebig erfolgen. Die Positionen 1 bis 5 sind nicht für die Angabe eines Programmcodes verfügbar, sondern dienen lediglich als Informationsstellen (z. B. für Änderungsmarkierungen).

Die Position 6 muss in jedem Fall leer sein, während Position 7 für zusätzliche Compiler-Direktiven reserviert ist. Die Free-Format-Anweisungen enden in jedem Fall mit der Stelle 80 des Quellcodes.

Nachfolgend findet sich ein Beispiel für die Verwendung des Start- und Ende-Blocks der Free-Format-Anweisung der ersten Generation. Wenn Sie das neue Free-Format verwenden (nach der Version 7.1 und TR7), dann können Sie auf die hier aufgezeigten Start- und Ende-Kennzeichnungen verzichten.



Free-Start-Anweisung

Im vorhergehenden Beispiel finden sich einige Kommentarzeilen, eine Definitionsbestimmung und in Zeile 001500 die Startanweisung für den Free-Format-Block.

Die Zeilennummerierung ist zudem Bestandteil des RPG-Programmcodes und dient der Orientierung.

Bei diesem RPG-Codebeispiel handelt es sich um ein kompilierfähiges Beispiel, das keine wirkliche Tätigkeit ausführt. Dies aber ist für den Compiler vorerst nicht von Bedeutung.

Ein wichtiger Hinweis zu Anfang der Behandlung der Rechenbestimmungen:

Ein Muss im Kodieren eines RPG-Programms ist die Verwendung des Anzeigers für das Programmende. Der Anzeiger *inlr wird auch im vorangegangenen Beispiel genutzt. Der Sonderwert *INLR ist innerhalb des RPG bekannt und einer seiner wenigen Bestandteile, die nicht explizit definiert werden müssen. Wichtig ist, dass beim Kodieren des RPG-Programms in jedem Fall an der Ende-Stelle des RPG-Programmcodes *INLR aktiviert wird. Dies geschieht in dem Beispiel mit der Zeile 001800 *inlr = *ON.

Um ans Kodieren anzuschließen, ist Folgendes zu erwähnen: Mit dem freien Format in RPG verbunden ist auch die Notwendigkeit, dass das Ende der Zeile markiert werden muss. Grundsätzlich setzt das Free-Format-RPG den Anwendungsentwickler in die Lage, einen Code einzugeben, der im Kontext zusammengehörig ist und, würde es der Editor erlauben, in einer Zeile eingegeben werden könnte. Da die maximale Spaltenanzahl bei 80 bzw. 112 liegt, kann es gelegentlich vorkommen, dass solche Codeeingaben sich über mehrere Zeilen erstrecken. Das Ende einer Zeile oder des auf diese Weise auf mehrere Zeilen

verteilten Codes muss zwingend mit dem Semikolon (;) markiert werden! Jedes vergessene Semikolon wird unbarmherzig mit einem Wandlungsfehler quittiert.

Innerhalb des so definierten Free-Format-Blocks können sowohl alle verfügbaren Free-Format-Anweisungen als auch alle unterstützten Built-In Functions verwendet werden, wobei auf Letztere an späterer Stelle zurückgekommen wird.

Allgemein sollte sich ein guter Anwendungsentwickler von Anfang an daran gewöhnen, gewisse Standards und Regeln zu definieren und diese zu befolgen, um sich zum einen selbst Erleichterungen zu verschaffen und es zum anderen Kollegen zu ermöglichen,, den Programmcode einfacher verstehen und warten zu können. Deshalb gilt: Gewöhnen Sie sich von Anfang an einen guten und kontinuierlichen Programmierstil an!

Ein guter Stil ist z. B. der Beginn der „äußeren“ Programmlogik an Stelle 8 des Quellcodes, während die „innere“ Logik zwei Stellen weiter rechts beginnt. Solche Einrückungen erleichtern das Lesen eines RPG-Codes deutlich. Auf diese Weise lassen sich Abhängigkeiten und Blöcke optisch perfekt darstellen. Dazu ein Beispiel in der nachfolgenden Abbildung.

LFPBBKM.RPGLE

```
Zeile 175     Spalte 1     Ersetzen
     ...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+
012602       //*****************************************************************
012603
012604       Begsr sr_LesBestKopf;
012605
012606        BBEK01_k.EKFirm = BBKM01_I.BBKMFirm;
012607        BBEK01_k.EKBejj = BBKM01_I.BBKMBeJJ;
012608        BBEK01_k.EKBeNr = BBKM01_I.BBKMBeNR;
012613
012614        setll %kds(BBEK01_K:3) BBEK01;
012615        reade %kds(BBEK01_K:3) BBEK01 BBEK01_I;
012616
012617        dow not %eof(BBEK01);
012618
012619          // Lieferantenstamm einlesen
012620          LIEF01_K.LfFirm = BBEK01_I.EkFirm;
012621          LIEF01_K.LfLfNr = BBEK01_I.EkLfNr;
012622
012623          chain %kds(LIEF01_K:2) LIEF01 LIEF01_I;
012624          if %found(LIEF01);
012625          endif;
012626
012627         // exsr sr_LesBestPos; // nach Telko 24.11.15 auf BBUP umgebaut
012628          exsr sr_LesBestPosU;
012629          w_KzVerarb = '*';
012630
012631         reade %kds(BBEK01_K:3) BBEK01 BBEK01_I;
012632        enddo;
012633
012634         if w_KZKopf = *blank and w_KzVerarb <> *Blank;
012635          exsr sr_WriteMatKopf;
012636          w_KZKopf = '*';
012637         endif;
```

Beispiel für Einrückungen

3.2.4.5.1 Groß- und Kleinschreibung

Es gibt mit RPG keine Regel, nach der Variablennamen, Operationscodes oder Kommentare groß geschrieben werden müssen. Einige Programmierer haben sich selbst einen gewissen „Standard“ angewöhnt: das Großschreiben des ersten Zeichens von Variablennamen bzw. deren Einzelbestandteilen, während alle anderen Buchstaben als Kleinbuchstaben fortgesetzt werden. Diese Form des Kodierens (Kamelschreibweise) kann auch in RPG genutzt werden und erleichtert das Arbeiten mit dem Code. Letztlich bleibt es Ihnen überlassen, wann und ob Sie die Groß- bzw. die Kleinschreibung vorziehen. Der Compiler wandelt in jedem Fall für die interne Verarbeitung den von Ihnen eingegebenen Code zunächst in Großbuchstaben um. Das dient allerdings nur der internen Verarbeitung und hat für den Anwendungsentwickler keine Bedeutung.

3.2.4.5.2 Wertzuweisungen

Bevor mit der Logik in den Rechenbestimmungen gestartet wird, sollen zunächst einige weitere Grundlagen betrachtet werden.

Ein wichtiges Werkzeug in einer Programmiersprache ist das Zuweisen von Werten zu Variablen, Feldinhalten und anderen Bestandteilen.

RPG nutzt für die Wertzuweisung unabhängig von der Art des Ziels das Gleichheitszeichen „=“.

Dabei wird dem links neben dem Gleichheitszeichen stehenden Feld der Wert des rechts neben dem Gleichheitszeichen befindlichen Feldes oder der Konstanten zugewiesen.

Der Editor erlaubt zwar unterschiedliche Formen oder auch Schreibweisen in Bezug auf die Zuweisungen, die einfachste ist aber die Schreibweise, bei der die Zuweisung über das Gleichheitszeichen erfolgt. Dabei ist der Name der Variablen links vor dem Gleichheitszeichen anzugeben und der Zuweisungswert nach dem Gleichheitszeichen.

Bei der Wertzuweisung ist darauf zu achten, dass der Belegungswert korrekt kodiert wird. Alphanumerische Werte sind grundsätzlich in Hochkomma einzugeben. Hier muss auch auf die korrekte Groß- bzw. Kleinschreibung geachtet werden!

Bevor die eigentliche Zuweisung erfolgt, kümmert sich RPG eigenständig um das Initialisieren des Feldes, sodass nicht etwa zuvor eingestellte Werte das Ergebnis verfälschen. Ebenso werden dabei die Länge des Feldes nach den Standardvorgaben und der Initialisierungswert in Abhängigkeit des Typs der Variablen berücksichtigt.

3.2.4
Seite 72

Zunächst folgt ein Beispiel für eine Wertzuweisung.

RPG916.RPGLE

```
Zeile 30      Spalte 57    Einfügen
      ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....
000100     C*****************************************************************
000200     C*
000300     C*  RPG Free Beispiel
000400     C*
000500     C* Copyright (c) Joerg Zeig / ZeBIS 2013
000600     C*****************************************************************
000700     D var             S              1A
000701     D var1            S             10A   inz
000702     D Numm1           S              5  0 inz
000703     D Numm2           S              5  2 inz
001400
001500      /FREE
001600        var = *blank;    // Feld initialisieren Leerwert zuweisen
001601        var = *blanks;   // Feld initialisieren Leerwert zuweisen
001602        var = ' ';       // Feld initialisieren Leerwert zuweisen
001603        var = 'A';       // Wert A wird Variablen zugewiesen
001604
001605        var1 = *blank;   // Feld initialisieren Leerwert zuweisen
001606        var1 = 'ABCDEF';// Wert ABCDEF zuweisen
001607
001609        numm1 = *zero;   // Feld initialisieren Nullwert zuweisen
001610        numm1 = *zeros;  // Feld initialisieren Nullwert zuweisen
001611        numm1 = 15;      // Feld Wert zuweisen
001612
001613        numm2 = *zero;   // Feld initialisieren Nullwert zuweisen
001614        numm2 = 15,2;    // Feld Wert zuweisen
001615
001700
001800        *inlr = *on;
```

Wertzuweisungen

In diesem Beispiel ist eine Reihe unterschiedlicher Wertzuweisungen vorzufinden. Diese variieren in Abhängigkeit einiger unterschiedlicher Variablen, die zuvor in den Definitionsbestimmungen hinterlegt wurden. Hier eine Übersicht über die Variablen:

- Var Alphanumerische Variable, Länge: 1
- Var1 Alphanumerische Variable, Länge: 10
- Numm1 Numerische Variable, Länge: 5,0
- Numm2 Numerische Variable, Länge: 5,2

Exemplarisch wurden sowohl zwei alphanumerische als auch zwei numerische Variablen verwendet.

Im ersten Zuweisungsblock für die Variable „var“ finden sich zwei Beispiele für das beeinflusste Initialisieren, das über die Zuweisungen „*blank“ und „*blanks“ erfolgt.

```
var = *blank;    // Feld initialisieren Leerwert zuweisen
var = *blanks;   // Feld initialisieren Leerwert zuweisen
```

Es ist unerheblich, welche der beiden Initialisierungsformen für ein alphanumerisches Feld genutzt werden, der Effekt ist derselbe – das Feld oder die Variable wird geleert.

Um auf die unterschiedlichen Schreibweisen bei der Wertzuweisung zurückzukommen: Im Beispiel wurde das Gleichheitszeichen als Auslöser für die Wertzuweisung genutzt. Möglicherweise ist manchmal eine andere Form der Schreibweise, bei der der Operant „EVAL“ genutzt wird, vorzufinden. Aus früheren RPG-Varianten ist die Zuweisung mittels EVAL bekannt, und sie musste bei der Wertzuweisung im Fix-Format zwingend genutzt werden. Einige Anwendungsentwickler verwenden auch heute noch EVAL im Free-Format.

Nun folgt das Beispiel noch einmal, nur diesmal mit der Verwendung des Zuweisungsoperanten „EVAL“.

```
001616         eval var = *blank;    // Feld initialisieren Leerwert zuweisen
001617         eval var = *blanks;   // Feld initialisieren Leerwert zuweisen
001618         eval var = ' ';       // Feld initialisieren Leerwert zuweisen
001619         eval var = 'A';       // Wert A wird Variablen zugewiesen
001620
001621         eval numm2 = *zero;   // Feld initialisieren Nullwert zuweisen
001622         eval numm2 = 15,2;    // Feld Wert zuweisen
001623
```

EVAL-Beispiel

Die EVAL-Beispiele liefern dasselbe Ergebnis, wie es bereits bei der Zuweisung mittels des Gleichheitszeichens zu sehen war. Es handelt sich letztlich um eine doppelte Definition einer Zuweisung, die gelegentlich Anwendung findet, und es bleibt dem Anwender überlassen, welche der beiden Formen er nutzen will.

Abschließend seien einige weitere Zuweisungsbeispiele aufgeführt, bei denen einer Variablen der Wert einer anderen Variablen zugewiesen und bei einer numerischen Variablen eine Zuweisung mit einer Addition durchgeführt wird.

```
var1 = var;          // Zuweisung eines Variablenwertes
Numm1 = Numm3 + 2;   // Zuweisung eines numerischen Wertes
```

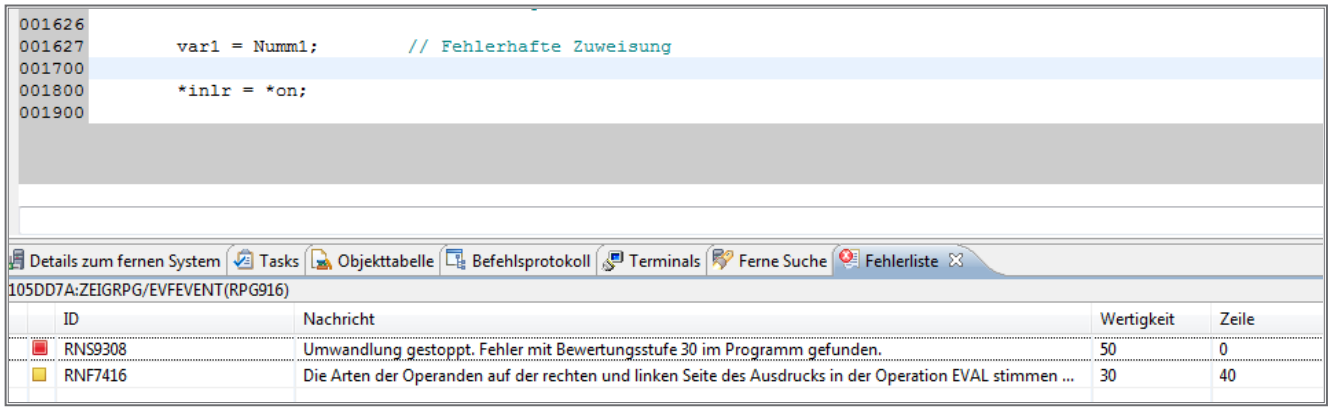
Weitere Zuweisungsbeispiele

3.2.4
Seite 74

Grundsätzlich ist bei der Zuweisung zu beachten, dass die Arten der Variablen und der zugewiesenen Werte passen. Das bedeutet z. B., dass nicht ohne Weiteres ein numerischer Wert in eine alphanumerische Variable eingestellt werden kann. Hier muss sich bei Bedarf zusätzlicher Funktionen bedient werden. Sollten Sie versuchen, eine fehlerhafte Zuweisung zu kodieren, dann erkennen Sie den Fehler erst beim Kompilieren, nicht aber beim Editieren der Quelle.

Ein Beispiel einer fehlerhaften Zuweisung wäre eine Zuweisung eines numerischen Werts in einen alphanumerischen Wert. Dazu sollen die beiden Beispielvariablen „var1“ und „Numm1“ herangezogen werden.

Das Kodieren selbst weist nicht auf die ungültige Zuweisung hin. Erst beim Kompilieren kommt es zum Wandlungsfehler RNF7416. Ein Blick auf die folgende Abbildung verrät dies.



Wandlungsfehler mit fehlgeschlagener Zuweisung

Im Klartext zum Fehler heißt es:

RNF7416: Die Arten der Operanden auf der rechten und linken Seite des Ausdrucks in der Operation EVAL stimmen nicht überein.

Beachten Sie auch hier den Fehlertext. Hier wird von der Operation „EVAL“ gesprochen. Diese war in der Zuweisung nicht angegeben – aber zuvor konnte gesehen werden, dass das Weglassen der Anweisung „EVAL“ lediglich eine Verkürzung der Zuweisung bedeutet.

Besondere Form der Wertzuweisungen

Auch wenn hier den Zuweisungen in RPG vorgegriffen wird – natürlich bietet RPG Möglichkeiten der Zuweisung verschiedenartiger Variablen und Werte, sofern dies nicht zu einem Fehler führt. Diese besondere Form der Zuweisung wird in RPG mithilfe von sogenannten Built-In Functions (kurz: BIF) umgesetzt.

Die Notwendigkeit, unterschiedliche Variablenarten miteinander zu kombinieren bzw. unterschiedliche Werte zuzuweisen, tritt hin und wieder auf. Mal sind es unterschiedliche Systeme und variierende Datendefinitionen, mal sind es Notwendigkeiten, die im Zusammenhang mit Datumswerten stehen.

Das bedeutet in diesem Fall (der Zuweisung eines numerischen Werts in einen alphanumerischen Wert): Verwendung der BIF „%editc“ oder „%Char“.

Wie diese eingesetzt werden, soll im Folgenden gezeigt werden.

Die BIF sind eine Erweiterung in RPG, die IBM bereits vor einigen Jahren gestartet hat. Während zu Beginn der RPG-Einführung viele Teilschritte in der Programmierung aufwendig kodiert werden mussten, wurde bei IBM die Notwendigkeit erkannt, wiederkehrende Funktionen und Erleichterungen in der RPG-Programmierung bereitzustellen. Die ersten BIFs wurden daraufhin eingeführt.

Hier sei der Einsatz einer ersten BIF zusammen mit der Konvertierung aufgezeigt:

%CHAR ist die BIF, mit der ein Ausgangswert in einen alphanumerischen Wert konvertiert werden kann.

Bezogen auf hiesiges Beispiel muss die Kodierung wie folgt vorgenommen werden: Name der Zielvariablen, gefolgt von der BIF %CHAR, gefolgt vom umzusetzenden Wert oder der umzusetzenden Variablen (beide in Klammern geschrieben).

Nachfolgende Abbildung verrät mehr.

```
var1 = %char(Numm1); // Zuweisung mit Konvertierung
```

%Char-Verwendung

Angenommen, die Variable Numm1 beinhaltet die Zahl 2. In einer Debug-Anzeige wird der numerische Wert wie folgt dargestellt:

```
      33
      34          eval numm2 = *zero;  // Feld initialisieren Nullwert zuweisen
      35          eval numm2 = 15,2;   // Feld Wert zuweisen
      36
      37          var1 = var;          // Zuweisung eines Variablenwertes
      38          Numm1 = Numm3 + 2;   // Zuweisung eines numerischen Wertes
      39
      40          var1 = %char(Numm1); // Zuweisung mit Konvertierung
      41
      42          *inlr = *on;
      43
                                                                        Ende
Testhilfe
F3=Programm beenden     F6=Unterbrechungspunkt hinzufügen/löschen   F10=Step
F11=Variable anzeigen   F12=Wiederaufnehmen   F24=Weitere Tasten
NUMM1 = 00002.
```

Ausgangswert Numm1

Wie zu sehen ist, hat Numm1 den Wert 00002. Also ist gemäß der Felddefinition (5,0) der Feldwert fünfstellig, und der zuvor der Variablen zugefügte Wert 2 wird mit 00002 dargestellt.

Im Folgenden ist das Ergebnis der Zuweisung mittels %Char zu sehen:

```
      36
      37          var1 = var;          // Zuweisung eines Variablenwertes
      38          Numm1 = Numm3 + 2;   // Zuweisung eines numerischen Wertes
      39
      40          var1 = %char(Numm1); // Zuweisung mit Konvertierung
      41
      42
      43          *inlr = *on;
      44
                                                                        Ende
Testhilfe
F3=Programm beenden     F6=Unterbrechungspunkt hinzufügen/löschen   F10=Step
F11=Variable anzeigen   F12=Wiederaufnehmen   F24=Weitere Tasten
VAR1 = '2         '
```

Ergebnis der Zuweisung mit %Char

Die Variable var1 aus dem vorherigen Beispiel hat nach der Zuweisung mittels %char den alphanumerischen Wert 2.

Nun kann darüber diskutiert werden, ob das Ergebnis so korrekt umgesetzt ist. Denn: Ist eine Ziffer ohne oder mit führenden Nullen darzustellen?

Es kann also festgehalten werden:

- Mit %Char kann einem alphanumerischen Feld/einer alphanumerischen Variablen ein numerischer Wert zugewiesen werden.
- Führende Nullen aus dem numerischen Feld werden im alphanumerischen Feld entfernt und nicht umgesetzt.

Ob das die korrekte Umsetzung oder Anforderung ist, wird sicher von Fall zu Fall zu unterscheiden sein.

Deshalb sollte noch ein Blick auf eine weitere BIF geworfen werden, mit der bestimmte Zuweisungen durchgeführt werden können – die BIF %EDITC (Editieren eines Werts mithilfe eines Editier-Codes).

Wie der Name bereits besagt, beim Einsatz der BIF %EDITC werden Editiercodes genutzt – also von IBM bereitgestellte Regeln und Schablonen, mit denen Werte und Inhalte einer Variablen aufbereitet und dargestellt werden. Eine genaue Übersicht der Editiercodes ist auf den Webseiten der IBM niedergelegt.

Die Kodierung von %Editc ist vergleichbar mit dem Einsatz von %Char – mit einer wesentlichen Unterscheidung: Bei %Editc muss ein Parameter mitgegeben werden, der den Aufbereitungsschlüssel bzw. den Editiercode beinhaltet. Diese Angabe erfolgt nach einem Doppelpunkt in einfachen Hochkommazeichen. Dazu in folgendem Beispiel mehr.

var1 = %editc(Numm1:'X'); // Zuweisung mit Aufbereitung

```
001628
001629          var1 = %editc(Numm1:'X'); // Zuweisung mit Aufbereitung
001700
```

Beispiel für das Kodieren der BIF %Editc

Was bewirkt die BIF %Char in Verbindung mit dem X?

Dies erklärt sich in einer Debug-Sitzung.

```
     40          var1 = %char(Numm1); // Zuweisung mit Konvertierung
     41
     42
     43          *inlr = *on;
     44
                                                                  Ende
Testhilfe
F3=Programm beenden     F6=Unterbrechungspunkt hinzufügen/löschen   F10=Ste
F11=Variable anzeigen   F12=Wiederaufnehmen   F24=Weitere Tasten
VAR1 = '2          '
```

Debug-Anzeige

3.2.4
Seite 78

Mit dem Parameter X konnten die führenden Nullen im Ergebnisfeld entfernt werden.

Insbesondere die BIF %Editc bietet eine Vielzahl an Einsatzbereichen. Die Editiercodes, die auf den Webseiten der IBM genau beschrieben sind, zeigen dies auf.

3.2.5 RPG und Embedded SQL

Record Level Access (RLA) oder native I/O ist die klassische Zugriffsmethode, durch die mit Programmiersprachen wie RPG und Cobol Daten in physischen Dateien oder SQL-Tabellen erstellt, modifiziert und/oder gelöscht werden können.

Die meisten anderen Programmiersprachen, wie Java, C++ oder PHP, mit deren Hilfe immer häufiger auf die in der DB2 for i gespeicherten Daten zugegriffen wird, verwenden für den Zugriff fast ausschließlich SQL.

Embedded SQL erlaubt es, in RPG und Cobol SQL-Befehle direkt einzubinden und auszuführen. Datenbankenzugriffe können somit wahlweise mit native I/O oder mit SQL codiert werden.

3.2.5.1 Welche Vorteile bietet der Einsatz von embedded SQL?

- Bei Verwendung von embedded SQL kann der gleiche oder leicht modifizierte Quellcode sowohl in RPG als auch in Cobol bzw. in anderen Sprachen, wie Java oder PHP, die einen Zugriff auf die Daten mittels SQL zulassen, eingesetzt werden.
  Notwendige Änderungen und Erweiterungen können problemlos in den Quellcode, der in unterschiedlichen Programmiersprachen geschrieben sein kann, integriert werden. Anpassungen in RPG- oder Cobol-Quellen können selbst von Programmierern vorgenommen werden, die zwar SQL, nicht jedoch RPG oder Cobol beherrschen.
- Mit embedded SQL können alle Arten von SQL-Befehlen in RPG und Cobol eingebunden und ausgeführt werden. Neben den SQL Data Manipulation Language (DML) -Befehlen Insert, Update, Delete sowie dem Select-Statement können auch alle Befehle aus der SQL Data Definition Language (DDL), z. B. CREATE TABLE, verwendet werden.
- SQL liefert zurzeit ca. 170 skalare Funktionen, während in RPG im Vergleich dazu nur ungefähr 80 Built-In-Functions integriert sind. Mit Hilfe von embedded SQL können diese Funktionen direkt in RPG oder Cobol verwendet werden, ohne dass hierfür auf eine Tabelle/Datei/View zugegriffen werden muss.
- Mit einem einzigen SQL-Statement können Hunderte oder Tausende von Datensätzen eingefügt, geändert oder gelöscht werden.

3.2.5
Seite 2

- SQL liefert diverse Möglichkeiten, um physische Dateien, SQL-Tabellen oder SQL Views miteinander zu verknüpfen, z. B. INNER JOIN, OUTER JOIN, EXCEPTION JOIN, FULL JOIN und CROSS JOIN. SQL lässt nicht nur eine Verknüpfung von unterschiedlichen Dateien/Tabellen/Views zu, sondern erlaubt auch eine Verknüpfung der Dateien/Tabellen/Views mit sich selbst (SELF JOIN).
- Über UNION-, EXCEPT- und INTERSECT-Anweisungen können unabhängige SELECT-Befehle (z. B. Suche der Bestell-Nr. im Auftragskopf und Suche der Bestell-Nr. in der History-Datei) als Einheit ausgeführt werden.
- Verschachtelte SELECT-Statements sind ebenso zulässig wie Common Table Expressions (CTE = Temporäre Views, die nur für das aktuelle Select-Statement gelten).
- SQL bietet eine Reihe von Aggregatfunktionen, durch die Summen, Durchschnittswerte sowie Minimum und Maximum, aber auch die empirische Varianz einfach ermittelt werden können. Seit der Erweiterung in der GROUP BY-Anweisung in Release 6.1 können außerdem mit einem einzigen SELECT-Statement beliebige Zwischensummen und/oder die Gesamtsumme ermittelt und ausgegeben werden.
- Mit Hilfe von „Rekursiven Common Table Expressions“ (RCTE) oder der hierarchischen Query-Anweisung können Stücklisten und Organigramme aufgelöst und Fahr- oder Netzpläne analysiert werden.
- Dynamisches SQL erlaubt, dass SQL-Statements erst zur Laufzeit gebildet und sofort ausgeführt werden können.
- Mit embedded SQL kann in Verbindung mit Large Object Files (xLOB_File) direkt auf das Integrated File System (IFS) zugegriffen werden. Embedded SQL erlaubt nicht nur, diese Dateien zu lesen, sondern auch zu modifizieren oder gar zu erzeugen.
- Durch verstärkten Einsatz von (embedded) SQL kann Programmlogik in die Datenbank verlagert werden, d. h. ein großer Teil der Funktionalität kann in einigen wenigen SQL-Befehlen konzentriert werden. Der Quellcode kann dadurch z. T. beträchtlich reduziert werden.
- Anstatt komplexe SQL-Statements direkt im Quellcode zu hinterlegen, können diese auch in SQL-Views (Sichten) gespeichert werden. Im Idealfall wird der eigentliche Quellcode auf den Zugriff der View in Verbindung mit ein paar zusätzlichen WHERE-Bedingungen reduziert. Da eine SQL-View sowohl in Programmiersprachen wie RPG und Cobol, aber auch in JAVA, C++ und PHP und sogar in Tools wie Query/400 verwendet werden kann, wird auch der notwendige Testaufwand minimiert.

Muss der SQL-Code erweitert werden, muss lediglich die View angepasst und das Ergebnis aus einer Abfrage für die neue View geprüft werden. Der Quellcode muss unabhängig von der verwendeten Programmiersprache, in der die View eingebunden ist, weder angepasst werden noch müssen die (Service-)Programmobjekte neu erstellt werden.
Auf SQL-Views kann zwar auch mit native I/O zugegriffen werden, da Views jedoch immer ungeschlüsselt sind, d. h. die Datensätze keine bestimmte Reihenfolge haben, ist ihr Einsatz in Verbindung mit Record Level Access nur bedingt sinnvoll.

- Module und Programme, in denen embedded SQL verwendet wird, müssen bei der Erweiterung von Tabellen/Dateien nur dann neu erstellt werden, wenn SELECT * verwendet wurde.
  Wurden die Felder/Spalten dagegen explizit angegeben, und ist die Definition der verwendeten Spalten unverändert, ist eine Kompilierung nicht erforderlich.

- SQL (Standard Query Language) ist eine standardisierte Sprache zur Definition und Manipulation von Objekten und Daten in relationalen Datenbanken. Alle Hersteller von Datenbanken haben sich verpflichtet, diesen festgelegten Standard (aktuell SQL 2008) einzuhalten und abzudecken.
  Dabei ist jedoch anzumerken, dass es bislang nur eine einzige Datenbank gibt, die den Standard zu 100 % abdeckt und auch hier sind einige Features noch nicht voll ausgebaut. Diese Datenbank ist DB2 for i.
  Außerdem lässt auch ein Standard gewisse Interpretationsmöglichkeiten zu.
  Durch den Einsatz von Middleware ist es bereits heute möglich, mittels SQL auf Daten in anderen Datenbanken zuzugreifen.
  Die Verwendung von (embedded) SQL ist die Voraussetzung für den Zugriff auf andere Datenbanken. Bedingt durch den SQL-Standard können die gleichen, allenfalls leicht modifizierte SQL-Befehle in allen Datenbanken eingesetzt werden. Eine Migration kann also mit minimalem Aufwand bewerkstelligt werden.

Ergänzung 21/2012

3.2.5.2 Embedded SQL und Query-Optimierung

Bei der Verwendung von native I/O gibt der Programmierer den gewünschten Zugriffspfad vor. In RPG z. B. wird die entsprechende logische Datei (oder der SQL-Index) in den F-Bestimmungen angegeben und auch verwendet.

Nicht so bei der Verwendung von (embedded) SQL! Hier entscheidet der Query Optimizer, ob und welche Zugriffspfade verwendet werden.

In SQL-Statements können zwar DDS-beschriebene logische Dateien angegeben werden, der Query Optimizer muss in einem solchen Fall jedoch zunächst die DDS-Beschreibung analysieren und die Abfrage basierend auf den physischen Dateien/SQL-Tabellen neu schreiben. Aus der DDS-Beschreibung werden nur die Feldauswahl, JOIN- und SELECT/OMIT-Anweisungen übernommen. Die in der logischen Datei hinterlegten Schlüsselinformationen werden ignoriert. Die eigentliche Optimierung beginnt erst, nachdem die SQL-Abfrage überarbeitet wurde. Zu diesem Zeitpunkt ist jedoch nicht mehr bekannt, dass eine logische Datei mit einem bestimmten Schlüssel vorgegeben wurde.

Vor Release 7.1 konnte die Analyse der DDS-beschriebenen logischen Dateien nur von der alten Classic Query Engine (CQE) ausgeführt werden. Wurde also eine DDS-beschriebene logische Datei in einem SQL-Statement angegeben, so wurde diese Abfrage gnadenlos von der CQE ausgeführt, was einen Performanceverlust von bis zu 15 % bedeutet.

Anmerkung:

DDS-beschriebene logische Dateien sollten aus Gründen der Performance nicht in SQL-Abfragen verwendet werden.

Mit Release V5R2 wurde eine neue, wesentlich flexiblere und performantere SQL Query Engine (SQE) eingeführt, die schrittweise mehr und mehr Funktionalität der CQE übernommen hat. Mit Release 6.1 war die SQE komplett eingeführt. Beide Query Engines existieren jedoch weiterhin nebeneinander und der Query Dispatcher entscheidet auch weiterhin, von welcher Query Engine ein SQL-Statement ausgeführt wird. Man sollte allerdings beachten, dass neue Features in der DB2 for i und in SQL für die SQE optimiert sind und z. T. nur von ihr ausgeführt werden können.

Die Query-Optimierung ist ein „kostenbasiertes" System, d. h. die vorhandenen Zugriffswege (in DDS-beschriebenen logischen Dateien und SQL Indices) und die in den Statistiken gesammelten Informationen über die aktuellen Datenkonstellationen werden miteinander verglichen und bewertet. Die günstigste Kombination erhält den Zuschlag. Dabei muss auch berücksichtigt werden, dass basierend auf der Datenzusammensetzung ein Zugriffweg, der heute noch als optimal angesehen wurde, morgen nicht mehr verwendet wird. An dieser Stelle muss die Performance geprüft und müssen ggf. neue Zugriffswege angelegt werden.

Somit stellt sich die Frage, wie kann man sicherstellen, dass SQL-Abfragen möglichst performant ausgeführt werden?

Tatsächlich gibt es zwei Möglichkeiten, auf den Query Optimizer Einfluss zu nehmen:

- **Durch die Codierung der SQL-Statements**
  Unterschiedliche Schreibweisen des gleichen SQL-Statements können dazu führen, dass der Optimizer in einem Fall einen Zugriffsweg verwenden kann, während im anderen Fall die Tabelle komplett gelesen werden muss.
- **Durch die Bereitstellung der optimalen Zugriffswege**
  Im System i Navigator sind einige Tools zur Unterstützung der Performance-Analyse integriert, z. B. Visual Explain, Index Advisor, Index Evaluator.

Fazit:

Mit embedded SQL kann der Programmier- und Testaufwand beträchtlich reduziert werden. Man muss jedoch im Vergleich zu native I/O Zeit in die Performance-Optimierung investieren. Die Performance-Analyse und -Optimierung kann jedoch von einem einzigen SQL-Spezialisten vorgenommen werden, der auch die benötigten Views in der richtigen Schreibweise anlegt. Die Programmierer verwenden lediglich die fertigen Views.

Ergänzung 21/2012

3.2.5.3 Voraussetzungen für embedded SQL

Um RPG und COBOL-Programme mit embedded SQL erstellen zu können, müssen das Lizenzprodukt IBM® DB2 Query Manager and SQL Development Kit for i sowie die Sprachen-Compiler installiert sein.

In diesem Lizenzprodukt sind neben dem Query Manager, dem interaktiven SQL und dem SQL-Interface für REXX auch die SQL-Precompiler für RPG, Cobol, PL/I, C und C++ enthalten.

Der SQL-Precompiler umfasst die folgenden Kompilierungsbefehle:

- **CRTSQLCI** Erstellen SQL ILE C Object (= Program, Modul, Service-Program)
- **CRTSQLCPPI** Erstellen SQL ILE C++ Object
- **CRTSQLCBL** Erstellen SQL Cobol OPM Program
- **CRTSQLCBLI** Erstellen SQL ILE Cobol Object
- **CRTSQLPLI** Erstellen SQL PL/I Program
- **CRTSQLRPG** Erstellen SQL RPG OPM Program
- **CRTSQLRPGI** Erstellen SQL ILE RPG Object

Anmerkung:

Zur Ausführung von Programmen, die embedded SQL enthalten, ist das SQL Development Kit for i nicht erforderlich. Daher können z. B. RPG-Programme mit embedded SQL auf einer Entwickungsmaschine, auf der SQL installiert ist, erstellt und auf einer Maschine ohne SQL ausgeführt werden.

3.2.5.3.1 Quellen-Teildateien mit embedded SQL

Teildateien mit Quellcode, in dem embedded SQL-Statements hinterlegt werden, müssen als solche über die Teildateiart wie folgt ausgewiesen werden:

- **SQLCLE** ILE C mit embedded SQL
- **SQLCPPLE** ILE C++ mit embedded SQL
- **SQLCBL** OPM Cobol mit embedded SQL
- **SQLCBLLE** ILE Cobol mit embedded SQL
- **SQLPLI** PLI mit embedded SQL
- **SQLRPG** RPGIII mit embedded SQL
- **SQLRPGLE** ILE RPG mit embedded SQL

3.2.5.3.2 Codierung von SQL-Befehlen im Quellcode

Innerhalb des Quellcodes eingebettete SQL-Statements müssen in allen Programmiersprachen, die embedded SQL unterstützen, mit EXEC SQL gekennzeichnet werden. Beendet werden die SQL-Statements in den einzelnen Sprachen unterschiedlich.

In RPG können embedded SQL-Statements sowohl im fixen Format als auch im Free Format hinterlegt werden.

3.2.5.3.2.1 Codierung von SQL-Befehlen im fixen RPG-Format

Bei Einbindung im fixen RPG-Format beginnt ein SQL-Statement immer in Position 7 mit **/EXEC SQL.** Der eigentliche SQL-Befehl kann in der gleichen Zeile – nach EXEC SQL – beginnen. Den Beginn in die nächste Zeile zu verlagern, ist jedoch ebenfalls zulässig.

Der SQL-Befehl muss im fixen Format mit einer Endezeile mit **/END-EXEC** ab Position 7 abgeschlossen werden.

Zwischen /EXEC SQL und /END-EXEC können beliebig viele Zeilen, die den SQL-Code enthalten, eingefügt werden. Diese Zeilen müssen mit C+ in Position 6 und 7 gekennzeichnet werden. Der SQL-Code kann zwischen Position 8 und 80 frei codiert werden.

Zwischen /EXEC SQL und /END-EXEC kann immer nur **genau ein** SQL-Statement angegeben werden. Innerhalb der gleichen Quelle können jedoch beliebig viele SQL-Statements hinterlegt werden, jeweils beginnend mit /EXEC SQL und endend mit /END-EXEC.

Im folgenden Beispiel werden zwei SQL-Statements im fixen RPG-Format eingebunden. Im ersten Fall wird ein UPDATE-Statement hinterlegt, das erst in der Folgezeile beginnt. Im zweiten Fall wird der komplette DELETE-Befehl in der ersten Zeile angegeben und es werden keine Folgezeilen verwendet.

```
C/EXEC SQL
C+ Update MySchema/MyLib
C+ Set MyDate = Current_Date
C+ Where MyStatus = 'Erledigt' and MyDate is NULL
C/END-EXEC

C/EXEC SQL Delete From QTEMP/MyWrkTbl
C/END-EXEC
***** Quelle EMBSTMT
```

Einbindung von SQL-Befehlen in Fixed-Format RPG

3.2.5.3.2.2 Codierung von SQL-Befehlen in Free-Format RPG

In Free-Format RPG muss ein SQL-Befehl mit **EXEC SQL** beginnen. EXEC SQL kann an jeder beliebigen Position zwischen 8 und 80 angegeben werden.

In Free-Format RPG endet ein eingebetteter SQL-Befehl wie jeder andere Free-Format RPG-Befehl mit **Semi Colon (;).** Die Angabe von END-EXEC, die im fixen Format notwendig ist, entfällt.

Das eigentliche SQL-Statement kann nach EXEC SQL beginnen und über beliebig viele Zeilen fortgesetzt werden. Eine spezielle Kennzeichnung der Folgezeilen ist nicht erforderlich. Wichtig ist allerdings, dass der SQL-Code nur zwischen Position 8 und 80 angegeben werden darf.

Innerhalb der gleichen Quelle können beliebig viele embedded SQL-Befehle hinterlegt werden, jeweils beginnend mit EXEC SQL.

Im folgenden Beispiel werden zwei SQL-Statements in Free-Format RPG eingebunden. Im ersten Fall beginnt das eigentliche SQL-Statement in der Folgezeile, während im zweiten Fall das komplette SQL-Statement hinter dem EXEC SQL angegeben wurde. EXEC SQL wurde an unterschiedlichen Positionen codiert.

```
/Free
   EXEC SQL
        Update MySchema/MyLib
           Set MyDate = Current_Date
         Where MyStatus = 'Erledigt' and MyDate is NULL;

        EXEC SQL  Delete From QTEMP/MyWrkTbl;
/End-Free
****** Quelle EMBSTMT
```

Einbinden von SQL-Befehlen in Free-Format RPG

3.2.5.3.2.3 Codierung von SQL-Befehlen in Cobol

Jedes SQL-Statement, das in ein Cobol-Programm eingebunden wird, beginnt mit EXEC SQL, und mit END-EXEC. endet es. EXEC SQL und END-EXEC. müssen jeweils in einer eigenen Zeile angegeben werden.

Das eigentliche SQL-Statement wird zwischen den beiden EXEC-Zeilen angegeben. Für embedded SQL-Statements gelten die gleichen Fortsetzungsregeln wie für die normalen Cobol-Statements.

Im folgenden Beispiel werden zwei SQL-Statements in Cobol hinterlegt.

```
EXEC SQL
     Update MySchema/MyLib
        Set MyDate = Current_Date
      Where MyStatus = 'Erledigt' and MyDate is NULL
END-EXEC.

EXEC SQL
     Delete From QTEMP/MyWrkTbl
END-EXEC.
```

Einbinden von SQL-Befehlen in Cobol

3.2.5.3.2.4 Kommentare in (embedded) SQL-Befehlen

Innerhalb der SQL-Befehle können auch Kommentare hinterlegt werden. Zum Einbinden von Kommentaren gibt es mehrere Möglichkeiten:

- **Stern (*) auf Position 7** — Gilt nur für Fixed-Format RPG. Die komplette Zeile innerhalb des SQL-Statements wird auskommentiert.
- **Zwei Minuszeichen --** — Der auf das doppelte Minuszeichen folgende Text innerhalb der gleichen Zeile wird auskommentiert. Anweisungen, die vor dem doppelten Minuszeichen in der gleichen Zeile codiert sind, werden nicht auskommentiert.
- **/* Kommentar */** — Nur der zwischen /* und */ eingebettete Text wird als Kommentar behandelt. Anweisungen vor und nach /* und */ innerhalb der gleichen Zeile werden nicht auskommentiert.

Im folgenden Beispiel ist jeweils die Information MyMenge > 0 auskommentiert. Der erste Befehl wurde in fixem RPG-Format eingebunden. Durch den Stern (*) in Position 7 wird die komplette Zeile auskommentiert.

Die anderen beiden Statements sind in Free-Format RPG codiert. Durch die Angabe des doppelten Minuszeichens in dem ersten Free-Format-Befehl wird die komplette Zeile auskommentiert. In dem zweiten Befehl wird durch die Verwendung von /* und */ nur ein Teil der Anweisung auskommentiert, d. h. „and MyDate is NULL“ wird weiterhin ausgeführt.

```
C/EXEC SQL Update MySchema/MyLib
C+            Set MyDate = Current_Date
C+          Where     MyStatus =  'Erledigt'
C**** Kommentar   and MyMenge  >  0
C+             and MyDate   is NULL
C/END-EXEC

 /Free
     Exec SQL  Update MySchema/MyLib
                  Set MyDate = Current_Date
                Where     MyStatus =  'Erledigt'
      -- Kommentar    and MyMenge  >  0
                      and MyDate   is NULL;

      Exec SQL  Update MySchema/MyLib
                   Set MyDate = Current_Date
                 Where     MyStatus =  'Erledigt'
                  and /* MyMenge  >  0 and */  MyDate is NULL;
 /End-Free
 ****** Quelle EMBCOMM
```

Einbindung von Kommentaren in embedded SQL-Statements

3.2.5.3.3 Erstellung von Objekten mit embedded SQL

Programme, Module oder Serviceprogramme, die embedded SQL enthalten, müssen vor der eigentlichen Umwandlung durch den Language Compiler zunächst vorkompiliert werden.

Die Vorkompilierung wird durch die im SQL Developer Toolkit for i ausgelieferten Kompilierungsbefehle angestoßen.

3.2.5.3.3.1 Kompilierungsbefehle

Erstellt werden die Module, Programme und Serviceprogramme, die embedded SQL enthalten, mit einem der folgenden Befehle:

- **CRTSQLCI** Erstellen SQL ILE C Object (= Program, Modul, Service-Program)
- **CRTSQLCPPI** Erstellen SQL ILE C++ Object
- **CRTSQLCBL** Erstellen SQL Cobol OPM Program
- **CRTSQLCBLI** Erstellen SQL ILE Cobol Object
- **CRTSQLPLI** Erstellen SQL PL/I Program
- **CRTSQLRPG** Erstellen SQL RPG OPM Program
- **CRTSQLRPGI** Erstellen SQL ILE RPG Object

Erfolgt die Umwandlung über PDM (Program Development Management / Programmentwicklungsverwaltung) oder RDp (Rational Developer for RPG and COBOL), werden die entsprechenden Vorkompilierungsbefehle abhängig von der Teildateiart automatisch herangezogen und verwendet.

Für alle ILE-Programmiersprachen (RPG, Cobol, C, C++) gibt es jeweils nur einen einzigen Kompilierungsbefehl, mit dem sowohl Module als auch Programme und Serviceprogramme erstellt werden. Die Art des zu erstellenden Objekts wird über die Art der Kompilierung (**OBJTYPE**) im Kompilierungsbefehl vorgegeben.

Im folgenden Beispiel wird ein ILE RPG-Programm mit embedded SQL erstellt.

```
CRTSQLRPGI OBJ(MYLIB/MYPGM)
           SRCFILE(MYLIBSRC/QRPGLESRC)
           SRCMBR(MYPGM)
          OBJTYPE(*PGM)
```

Festlegen der Objektart für das zu erstellende Objekt mit embedded SQL

Die CRTSQLxxx-Befehle enthalten nicht alle Optionen, die z. B. in den CRTxxx-MOD (Modul erstellen) oder CRTBNDxxx (Binder-Programm erstellen) angegeben werden können. So gibt es z. B. keine Möglichkeit, die Aktivierungsgruppe festzulegen oder Binderverzeichnisse vorzugeben.

In RPG kann dieser Mangel durch die Angabe von entsprechenden Schlüsselworten in den **H-Bestimmungen** (z. B. ACTGRP oder BNDDIR) behoben werden.

Eine andere Möglichkeit besteht darin, solche Informationen im Vorkompilierungsbefehl über die Option **COMPILEOPT** (Kompilierungsoptionen) zu hinterlegen.

Im folgenden Beispiel werden die Kompilierungsoptionen DFTACTGRP und ACTGRP, die an den Befehl CRTBNDRPG (RPG Binder-Programm erstellen) übergeben werden sollen, in der Option COMPILEOPT hinterlegt.

```
CRTSQLRPGI OBJ(MYLIB/MYPGM)
           SRCFILE(MYLIBSRC/QRPGLESRC)
           SRCMBR(MYPGM)
           COMPILEOPT('DFTACTGRP(*NO) ACTGRP(*CALLER)')
```

Festlegen von zusätzlichen Kompilierungsoptionen

3.2.5.3.3.2 Kompilierung von Modulen/Programmen mit embedded SQL

Bei der Vorkompilierung erfolgt zunächst eine Syntaxprüfung der eingebetteten SQL-Befehle in Verbindung mit der Prüfung der in den SQL-Statements verwendeten (Host-)Variablen.

Sofern sowohl die Syntax als auch die Variablen-Definitionen korrekt sind, wird ein Duplikat der ursprünglichen Quelle in der Datei QSQLTEMP1 in der Bibliothek QTEMP erstellt. Die eingebetteten SQL-Befehle werden auskommentiert und durch API (Application Programming Interface) -Aufrufe ersetzt.

In RPG-Quellen werden die SQL Communications Area sowie weitere Variablen und Konstanten eingebunden, die von den aufgerufenen APIs benutzt werden. In anderen Programmiersprachen, z. B. Cobol, muss die SQLCA manuell über einen INCLUDE-Befehl in die Quellen integriert werden.

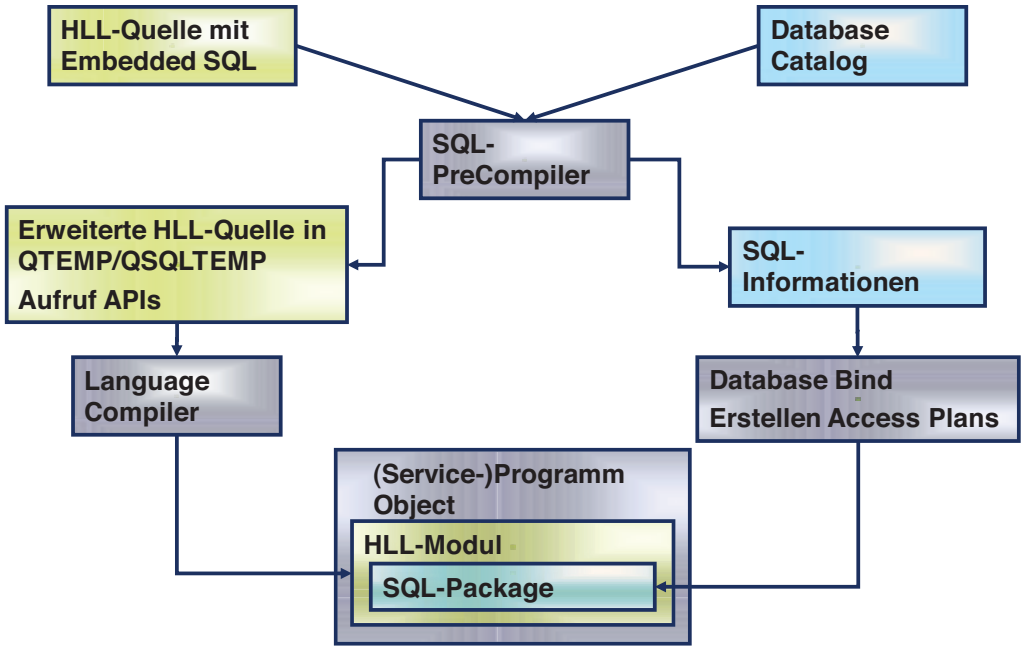
Der kopierte und modifizierte Quellcode wird an den entsprechenden Language-Compiler übergeben, d. h. abhängig von der Programmiersprache und der zu erstellenden Objektart wird entweder der Befehl CRTxxxMOD (RPG/Cobol/C/C++ Modul erstellen) oder CRTBNDxxx (RPG/Cobol/C/C++ Binder-Programm erstellen) aufgerufen.

Ergänzung 21/2012

3.2.5
Seite 14

Parallel zur Kompilierung werden die SQL/Datenbanken-Informationen analysiert und basierend auf den aktuellen Datenkonstellationen wird ein erster Zugriffsplan (Access Plan) ermittelt. Dieser Zugriffsplan wird als SQL Package in das Objekt integriert. In einem Zugriffsplan sind die Zugriffsmethoden (z. B. Index Access oder Table Scan), die Zugriffwege und die temporären Objekte beschrieben, die zur Ausführung des SQL-Statements erstellt werden müssen. Zur Laufzeit wird dieser Zugriffsplan zur Optimierung der SQL-Statements herangezogen, validiert und ggf. der verbesserte Zugriffsplan in das (Service-) Programm-Objekt zurückgeschrieben.

Das folgende Schaubild gibt einen Überblick über die Aktionen, die bei der Erstellung von Programmobjekten mit embedded SQL ausgeführt werden.



(Service-)Programmobjekte mit embedded SQL erstellen

3.2.5.3.3.3 SQL Communication Area (SQLCA)

Die SQL Communication Area (SQLCA) ist eine Datenstruktur, die in RPG vom SQL-Precompiler automatisch eingebunden wird.

Der folgende Quellcode zeigt die SQLCA, die vom SQL Precompiler in den ILE RPG-Quellcode integriert wird.

```
D SQLCA           DS
D  SQLCAID                       8A
INZ(X'0000000000000000')
D  SQLAID                        8A   OVERLAY(SQLCAID)
D  SQLCABC                      10I 0
D  SQLABC                        9B 0 OVERLAY(SQLCABC)
D  SQLCODE                      10I 0
D  SQLCOD                        9B 0 OVERLAY(SQLCODE)
D  SQLERRML                      5I 0
D  SQLERL                        4B 0 OVERLAY(SQLERRML)
D  SQLERRMC                     70A
D  SQLERM                       70A   OVERLAY(SQLERRMC)
D  SQLERRP                       8A
D  SQLERP                        8A   OVERLAY(SQLERRP)
D  SQLERR                       24A
D   SQLER1                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER2                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER3                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER4                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER5                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER6                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLERRD                     10I 0 DIM(6)OVERLAY(SQLERR)
D  SQLWRN                       11A
D   SQLWN0                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN1                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN2                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN3                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN4                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN5                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN6                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN7                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN8                       1A   OVERLAY(SQLWRN:*NEXT)
```

```
D   SQLWN9                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWNA                       1A   OVERLAY(SQLWRN:*NEXT)
D  SQLWARN                       1A   DIM(11) OVERLAY(SQLWRN)
D  SQLSTATE                      5A
D  SQLSTT                        5A   OVERLAY(SQLSTATE)
 ******* Quelle EMBSQLCA
```

SQL Communication Area für ILE RPG

In der SQLCA, die in RPG eingebunden wird, sind einige Datenstruktur-Unterfelder mehrfach, d. h. mit einem sechs Zeichen langen und einem längeren Namen (z. B. SQLCOD und SQLCODE) angelegt. Der Grund hierfür ist, dass embedded SQL auch schon in früheren RPG-Versionen eingesetzt werden konnte. In diesen Versionen konnten jedoch Datenstruktur-Unterfeld-Namen mit mehr als sechs Zeichen nicht definiert werden. Während die längeren Namen in allen anderen Programmiersprachen (z. B. Cobol und C) verwendet werden konnten, mussten für RPG die Namen gekürzt werden. Mit Release V5R3 wurden die längeren (Standard-)Namen in die RPG SQLCA integriert und die Datentypen angepasst, d. h. Integer (I) statt Binär (B) verwendet.

Anmerkung:

In neuen ILE RPG-Programmen mit embedded SQL sollten für die Prüfung grundsätzlich die längeren Namen verwendet werden.

In anderen Programmiersprachen, wie z. B. Cobol, muss die SQL Communication Area explizit über einen **INCLUDE-Befehl** in den Quellcode eingebunden werden.

Der folgende Quellcode zeigt den Auszug einer Kompilierungsliste für Cobol, nachdem der SQL Precompiler den INCLUDE-Befehl aufgelöst hat.

```
*****EXEC SQL
*****   INCLUDE SQLCA
*****END-EXEC.
01 SQLCA.
   05 SQLCAID PIC X(8). VALUE X"0000000000000000".
   05 SQLCABC PIC S9(9) BINARY.
   05 SQLCODE PIC S9(9) BINARY.
   05 SQLERRM.
      49 SQLERRML PIC S9(4) BINARY.
      49 SQLERRMC PIC X(70).
   05 SQLERRP PIC X(8).
```

```
05 SQLERRD OCCURS 6 TIMES
              PIC S9(9) BINARY.
   05 SQLWARN.
      10 SQLWARN0 PIC X.
      10 SQLWARN1 PIC X.
      10 SQLWARN2 PIC X.
      10 SQLWARN3 PIC X.
      10 SQLWARN4 PIC X.
      10 SQLWARN5 PIC X.
      10 SQLWARN6 PIC X.
      10 SQLWARN7 PIC X.
      10 SQLWARN8 PIC X.
      10 SQLWARN9 PIC X.
      10 SQLWARNA PIC X.
   05 SQLSTATE PIC X(5).
```

SQL Communication Area in Cobol einbinden

Die SQLCA wird von jedem eingebetteten SQL-Befehl (bzw. den APIs, die anstelle des SQL-Befehls ausgeführt werden) aktualisiert. Die SQLCA liefert vor allem Informationen darüber, ob ein SQL-Statement erfolgreich ausgeführt werden konnte oder fehlerhaft beendet wurde.

Im Fehlerfall bricht die Verarbeitung bei embedded SQL nicht ab, stattdessen wird ein entsprechender Fehlercode in der SQLCA ausgegeben.

Die folgenden Datenstruktur-Unterfelder der SQLCA liefern wichtige Informationen, die nach jedem SQL-Statement geprüft werden sollten:

- **SQLCODE**/SQLCOD
  SQLCODE < 0 ➜ Fehler
  SQLCODE = 100 ➜ Datensatz nicht gefunden
  SQLCODE > 0 und nicht 100 ➜ Warnung
  SQLCODE = 0 ➜ ordnungsgemäß ausgeführt
- **SQLSTATE**/SQLSTT
  Position 1-2 = 00 ➜ ordnungsgemäß ausgeführt
  Position 1-2 = 01 ➜ Warnung
  Position 1-2 = 02 ➜ Datensatz nicht gefunden
  Position 1-2 <> 00, 01, 02 ➜ Fehler
- **SQLERRMC**/SQLERM
  Im Fehlerfall werden in diesem Datenstruktur-Unterfeld die variablen Message-Texte ausgegeben.

- **SQLERRD(3)**/SQLER3
  In diesem Datenstruktur-Unterfeld wird die Anzahl hinzugefügter, geänderter oder gelöschter Zeilen nach einem Insert-, Update- oder Delete-Statement ausgegeben.
  Bei einem Multiple-Row-Fetch (Multiple Row Fetches) wird die Anzahl der en bloc gelesenen Zeilen/Datensätze in diesem Datenstruktur-Unterfeld ausgegeben.
- SQLERRD(5)
  Bei einem Multiple-Row-Fetch werden in diesem Datenstruktur-Unterfeld Informationen darüber ausgegeben, ob noch weitere Datensätze vorhanden sind.
  SQLERR(5) = 100 ➜ keine weiteren Zeilen/Datensätze

3.2.5.3.3.4 SQL-Befehl SET OPTION

In RPG werden für H-Bestimmungen Schlüsselworte bereitgestellt, über die Kompilierungsoptionen festgelegt werden können, z. B. ACTGRP oder ALWNULL. Diese Schlüsselworte werden jedoch nur von dem RPG-Compiler geprüft, nicht jedoch von dem SQL-Precompiler.

Vielfach ist es jedoch notwendig oder erwünscht, Umwandlungsoptionen auch in den CRTSQLxxx-Befehlen zu ändern. So ist z. B. der Unterlassungswert für Commit-Steuerung (COMMIT) in den CRTSQLxxx-Befehlen *CHG, d. h., es wird davon ausgegangen, dass die Dateien/Tabellen in einem Journal aufgezeichnet werden und Commitment Control verwendet wird. In vielen Unternehmen wird jedoch ohne Journaling und Commitment Control gearbeitet, d. h., die Option müsste standardmäßig auf *NONE geändert werden.

Anstatt die Kompilierungsoptionen jedes Mal direkt im Kompilierungsbefehl zu ändern, können Kompilierungsoptionen mit Hilfe eines **SET OPTION**-Befehls direkt im Source Code hinterlegt werden. Dabei ist jedoch Folgendes zu beachten:

- Das SET OPTION-Statement enthält Kompilierungsanweisungen, die für die komplette Quelle gelten. Deshalb darf nur **ein einziger SET OPTION**-Befehl mit allen Kompilierungsoptionen, die gesetzt werden sollen, angegeben werden.
- Das SET OPTION-Statement wird nur während der Vorkompilierung verwendet und zur **Laufzeit nicht ausgeführt.**

- Das SET OPTION-Statement muss **physisch als erstes SQL-Statement im Quellcode** hinterlegt werden. Dabei spielt es keine Rolle, ob das SET OPTION-Statement in der Main-Procedure oder in einer Sub-Procedure oder sogar in einer Quelle ohne Main-Procedure vor dem ersten P-Statement angegeben wird.

Empfehlung:

In RPG sollte das SET OPTION-Statement immer unmittelbar nach den globalen D-Bestimmungen eingefügt werden.

Über das SET OPTION-Statement können eine Vielzahl von Optionen angegeben werden. Die wichtigsten werden im Folgenden erklärt:

- **COMMIT** Commitment-Steuerung
  Der Unterlassungswert für die Option COMMIT im CRTSQLxxx-Befehl ist *CHG.
  Wird ohne Journaling und Commitment Control gearbeitet, muss diese Option auf *NONE gesetzt werden.

- **DATFMT** Datumsformat
  Der SQL-Precompiler generiert für jede Host-Variable, die in einem SQL-Befehl verwendet wird, ein zusätzliches Arbeitsfeld.
  Bei einem Datum wird das Datumsformat des Arbeitsfeldes weder über das Datumsformat der Host-Variablen bestimmt noch über das Datumsformat, das in den H-Bestimmungen hinterlegt ist. Stattdessen wird das Format verwendet, das im CRTSQLxxx-Befehl in der Option DATFMT angegeben wurde.
  Der Unterlassungswert für das Datumsformat im CRTSQLxxx-Befehl ist *JOB, d. h. das Datumsformat des aktuellen Jobs wird verwendet. Das Job-Datumsformat verwendet i. d. R. eine zweistellige Jahresangabe.
  Um Probleme bei der Übertragung aus der RPG-Host-Variablen in das SQL-Arbeitsfeld zu vermeiden, sollte ein Datumsformat mit einem vierstelligen Jahr, also *ISO, *EUR, *USA oder *JIS angegeben werden.

3.2.5
Seite 20

- **CLOSQLCSR**

Über die Option CLOSQLCSR wird angegeben, zu welchem Zeitpunkt die Cursor (Definition und Verwendung von Cursors) implizit geschlossen werden.
Beim impliziten Schließen werden die offenen Datenpfade (ODP) gelöscht. Beim nächsten Durchlauf muss ein **FULL OPEN** (komplette Optimierung) ausgeführt werden. Eine komplette Optimierung ist ein zeitaufwendiger Prozess, da der Zugriffsplan für die SQL-Befehle neu aufgebaut, zumindest jedoch validiert werden muss. Beim FULL OPEN müssen Zugriffswege bewertet sowie Zugriffsmethoden und temporäre Objekte definiert werden. Nachdem der Zugriffsplan erstellt oder aktualisiert wurde, müssen die darin beschriebenen temporären Objekte generiert und mit Daten gefüllt werden.
Sofern die Cursor nicht implizit geschlossen werden, können die offenen Datenpfade wiederverwendet werden. Beim nächsten Durchlauf wird nur ein **PSEUDO OPEN** ausgeführt, d. h., es müssen lediglich die Daten in den temporären Objekten aktualisiert werden.
Aus Gründen der Performance sollten die ODPs solange wie möglich offengehalten werden. Deshalb sollte der Unterlassungswert ***ENDACTGRP** nicht geändert werden, d. h., die ODPs werden erst beim Beenden der Aktivierungsgruppe gelöscht.
In Ausnahmefällen, wenn ein Programm z. B. beim Monatsabschluss nur einmalig ausgeführt wird, kann es jedoch erwünscht sein, dass die ODPs nach Beendigung des Moduls gelöscht werden. In diesem Fall sollte die Option CLOSQLCSR auf *ENDMOD gesetzt werden.

- **NAMING** Diese Option gibt an, ob für die auszuführenden SQL-Statements *SQL oder *SYSTEM-Naming verwendet wird.
  In embedded SQL wird standardmäßig *SYSTEM-Naming verwendet.
  Werden Datenbankenobjekte unter *SYSTEM-Naming qualifiziert angesprochen, so müssen Bibliothek und Objekt durch einen / (Schrägstrich) getrennt werden. Beim *SQL-Naming werden Bibliothek und Objekt durch einen . (Punkt) getrennt.
  Beim unqualifizierten Zugriff auf Dateien/Tabellen/Views wird beim *SYSTEM-Naming die Bibliotheksliste durchsucht. Beim *SQL Naming wird nur die Bibliothek mit dem Namen des angemeldeten Benutzers durchsucht oder die Bibliothek, die durch den SQL-Befehl SET CURRENT SCHEMA gesetzt wurde.

- **DFTRDBCOL** Standarddatensammlung / Bibliothek
  Diese Option ist nur bei Verwendung von ***SQL-Naming** interessant.
  Wird unter *SQL-Naming unqualifiziert auf Dateien/Tabellen/Views zugegriffen, wird nur **eine einzige Bibliothek** durchsucht.
  In der Option DFTRDBCOL kann die Datenbibliothek, die unter *SQL-Naming zur Laufzeit von **statischen SQL**-Statements verwendet werden soll, bereits zur Kompilierungszeit festgelegt werden.

- **DYNDFTCOL** Dynamische Standarddatensammlung / Bibliothek für dynamisches SQL
  Diese Option ist nur bei Verwendung von ***SQL-Naming** wichtig.
  In dieser Option kann festgelegt werden, ob unter *SQL-Naming zur Laufzeit für **dynamische SQL-Statements** die gleiche Datenbibliothek wie für statische SQL-Statements verwendet werden soll.
  Der Unterlassungswert ist *NO. Sofern unter *SQL-Naming für statische und dynamische SQL-Statements die gleiche Datenbibliothek verwendet werden soll, muss diese Option auf *YES gesetzt werden.

3.2.5
Seite 22

- **USRPRF** Gibt das Benutzerprofil an, das zur Laufzeit verwendet wird, um das Programmobjekt auszuführen.
  Der Unterlassungswert ist *NAMING, das bedeutet, dass für *SYSTEM-Naming der *USER und für *SQL-Naming der *OWNER verwendet wird.
  Das Benutzerprofil, das beim *SQL-Naming gesetzt wird, gilt nur für die statischen SQL-Statements.
  Das Benutzerprofil, das die dynamischen SQL-Statements ausführt, wird über die Option DYNUSRPRF gesetzt.
- **DYNUSRPRF** Dynamisches Benutzer-Profil
  Diese Option ist nur bei Verwendung von ***SQL-Naming** interessant.
  In der Option DYNUSRPRF wird angegeben, welcher Benutzer unter *SQL-Naming zur Laufzeit die dynamischen SQL-Statements ausführt.
  Der Unterlassungswert ist *USER, d. h., dynamische SQL-Statements werden vom aktuellen Benutzer ausgeführt. Die statischen SQL-Statements werden unter SQL-Naming per Default vom *OWNER ausgeführt.
  Sofern statische und dynamische SQL-Statements unter *SQL-Naming vom gleichen Benutzer ausgeführt werden sollen, muss entweder die Option DYNUSRPRF und/oder die Option USRPRF auf *OWNER oder *USER gesetzt werden.

Im folgenden Beispiel werden die Kompilierungsoptionen COMMIT, DATFMT und CLOSQLCSR gesetzt.

```
/Free
   Exec SQL   Set Option COMMIT    = *NONE,
                         DATFMT    = *ISO,
                         CLOSQLCSR = *ENDACTGRP;

   //Alle weiteren SQL-Statements
   //müssen nach dem SET OPTION-Statement codiert werden.
   ...
 /End-Free
 **** Quelle EMBSETOPT
```

Set Option-Statement

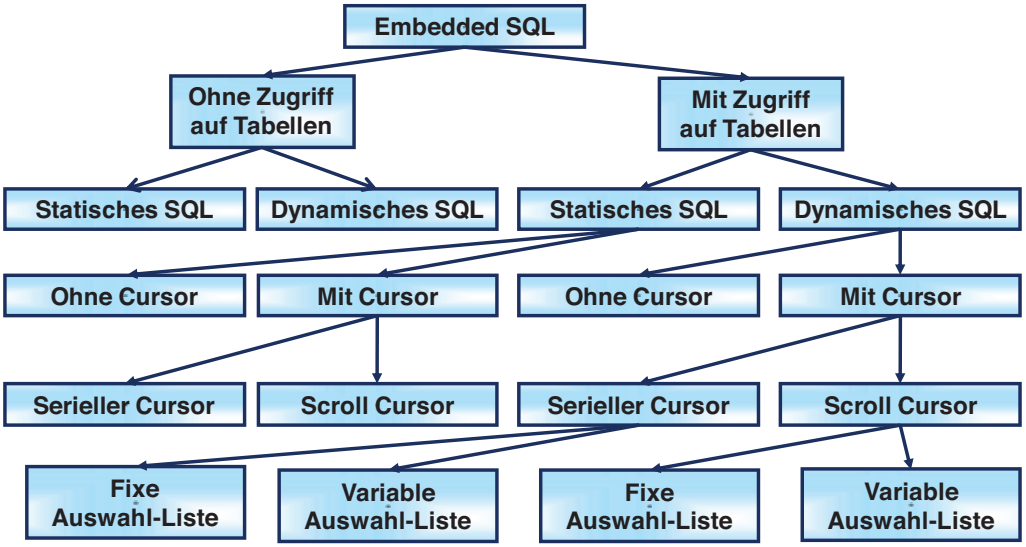
3.2.5.4 Verarbeitungsmethoden

In embedded SQL können alle Arten von SQL-Befehlen hinterlegt und ausgeführt werden.

Die Befehle aus der SQL Data Manipulation Language (DML), z. B. INSERT, können ebenso verwendet werden wie die Befehle aus der SQL Data Definition Language (DDL), z. B. CREATE TABLE, und es ist sogar möglich, Befehle aus der SQL-Programmierung einzubinden, z. B. SET.

Je nachdem, ob ein Datenbankenobjekt erstellt werden soll, ob nur eine skalare Funktion verwendet werden soll, ob ein Block von Daten mit einem einzigen SQL-Befehl modifiziert werden soll, ob nur ein einzelner Datensatz gelesen werden soll, ob eine Reihe von Datensätzen sequentiell verarbeitet werden soll, ob ein SQL-Statement bereits zur Kompilierungszeit bekannt ist oder erst zur Laufzeit aufbereitet werden kann, muss unterschiedlich vorgegangen werden.

Die folgende Abbildung gibt einen Überblick über die unterschiedlichen Verarbeitungsmethoden.



Embedded SQL – Verarbeitungsmethoden

- **Ohne Zugriff auf Tabellen**
  Beispiele für SQL-Befehle ohne Zugriff auf Tabellen, physische Dateien oder Views sind
  - Alle SQL Data Definition-Befehle (DDL) z. B. CREATE TABLE
  - Die SQL-Befehle GRANT und REVOKE zur Vergabe von Berechtigungen für Datenbankenobjekte
  - Die SQL-Befehle COMMIT und ROLLBACK für die Commitment-Steuerung
  - Der SQL-Befehl CALL zum Aufruf von Stored Procedures
  - Der SQL-Befehl SET aus der SQL-Programmierung zur Verwendung von SQL-skalaren Funktionen

  Die SQL-Befehle können statisch oder dynamisch aufbereitet werden. Bei statischem SQL ist der Befehl bereits zur Kompilierungszeit bekannt, während bei dynamischem SQL der Befehl erst zur Laufzeit aufbereitet wird.
- **Mit Zugriff auf Tabellen**
  Unter diesem Punkt werden alle SQL Data Manipulation Language-Befehle (DML) zusammengefasst.

  Über diese SQL-Statements werden i. d. R. mehrere Zeilen/Datensätze mit einem einzigen Befehl eingefügt, geändert oder gelöscht.

  Die SQL-Statements können statisch oder dynamisch aufbereitet werden.
- **Cursor-Verarbeitung**
  Cursor sind zur Verarbeitung von SELECT-Statements erforderlich, sofern mehr als eine Zeile/Datensatz (Result Set) ausgegeben wird und die Zeilen einzeln verarbeitet werden sollen.

  Wird nur eine einzelne Zeile ausgegeben, ist die Verwendung eines Cursors nicht zwingend erforderlich.

  Bei einem **seriellen Cursor** können die über das SELECT-Statement ausgegebenen Datensätze/Zeilen nur vorwärts gelesen und verarbeitet werden.

  Bei einem **Scroll Cursor** können die Datensätze/Zeilen vorwärts oder rückwärts gelesen werden. Ebenso ist eine Positionierung an den Anfang oder das Ende oder über eine relative Anzahl Zeilen innerhalb des Result Sets möglich.

Dynamisches und Statisches SQL sind auch bei der Cursor-Verarbeitung zulässig.

Beim dynamischen SQL wird das SELECT-Statement zur Laufzeit aufbereitet. Dabei muss unterschieden werden, ob immer mit der gleichen Spaltenauswahl oder mit variabler Spaltenauswahl gearbeitet wird.

Das Handling für eine variable Auswahlliste ist ungleich komplexer als die Verarbeitung einer fixen Spaltenliste.

3.2.5.5 Verwendung von Variablen

In statischen SQL-Statements können beliebige Variablen, Datenstrukturen und Datenstruktur-Unterfelder, die innerhalb des Quellcodes definiert werden, eingebunden werden.

Diese Variablen werden als **Host-Variablen** bezeichnet, während bei Datenstrukturen von **Host-Strukturen** geredet wird. Host-Variablen und Host-Strukturen müssen innerhalb des SQL-Statements durch einen **vorangestellten Doppelpunkt (:)** gekennzeichnet werden.

Host-Variablen werden dazu verwendet, Werte in SQL-Befehlen vorzugeben z. B.:

- Im SELECT-Befehl bei der Feldauswahl oder in den WHERE- oder HAVING-Anweisungen, um nur bestimmte Zeilen zu selektieren
- Im UPDATE- oder DELETE-Befehl, um die zu aktualisierenden oder zu löschenden Zeilen einzugrenzen
- Im UPDATE oder INSERT-Befehl, um neue Spalten-Werte zu setzen oder zu berechnen
- Im CALL-Befehl, um Parameter-Werte zu übergeben

Host-Variablen und Host-Strukturen können auch dazu verwendet werden, Ergebnisse aus einem SELECT-Statement zu empfangen.

Host-Variablen oder Unterfelder in Host-Strukturen sollten – soweit möglich – mit den gleichen, zumindest kompatiblen Datentypen und Längen definiert werden. Bei abweichenden Datentypen (z. B. alpha – numerisch) und/oder unterschiedlichen Längenangaben erfolgt eine interne Konvertierung. Bei Feldüberlauf gelten die üblichen (RPG-)Regeln, d. h., alphanumerische Werte werden abgeschnitten, bei numerischen Feldern oder Datums-Werten wird ein Fehler ausgegeben.

Bei den aus einer SQL-Abfrage ausgegebenen Spaltenwerten kann es sich auch um NULL-Werte handeln. NULL-Werte sind Werte, die außerhalb des gültigen Bereichs der Variablen liegen. RPG-Variablen können weder NULL-Werte empfangen noch verarbeiten. Das Schlüssel-Wort ALWNULL in den H-Bestimmungen und die Built-in-Function %NULLIND können nur in Verbindung mit native I/O verwendet werden. In SQL können NULL-Werte mit Hilfe von skalaren Funktionen (z. B. COALESCE) in Default-Werte konvertiert und ausgegeben werden. Es ist jedoch auch mit Hilfe von zusätzlichen Indikator-Variablen möglich, zu prüfen, ob in einer Spalte ein NULL-Wert ausgegeben wurde.

Vor Release 6.1 konnte der SQL-Precompiler lokalen Host-Variablen keine Prozedur oder Funktion zuordnen. Sofern lokale Variablen in unterschiedlichen Prozeduren mit dem gleichen Namen angelegt und identisch definiert waren (z. B. Artikel-Nr.), gab es keine Probleme. Bei abweichender Definition (z. B. LocText war in einer Prozedur als 50A und in der zweiten Prozedur mit 256C definiert) wurde vom SQL-Precompiler ein Kompilierungsfehler ausgegeben.

Host-Variablen sind nur in statischem SQL zulässig (das SQL-Statement ist bereits zur Kompilierungszeit bekannt). Bei dynamischem SQL, das zur Laufzeit aufbereitet wird, müssen die Variablenwerte innerhalb des SQL-Statements entweder aufgelöst oder durch Parameter-Marker gekennzeichnet werden.

3.2.5.5.1 Host-Variablen

Host-Variablen können innerhalb von SQL-Statements zum einen dazu verwendet werden, um Spaltenwerte vorzugeben, aber auch dazu, Ergebnisse aus SQL-Abfragen zu empfangen.

Anmerkung:

Host-Variablen können nicht dazu verwendet werden, um Dateien/Tabellen /Views oder Schemata/Bibliotheken zur Laufzeit variabel zu gestalten.

Host-Variablen können als **eigenständige Felder** in den D- oder C-Bestimmungen definiert werden.

Felder/Spalten aus Dateien, die in den **F-Bestimmungen** hinterlegt sind, können ebenfalls als Host-Variablen verwendet werden.

Auch **Parameter**felder, die im Procedure Interface oder in einer *ENTRY PLIST angegeben oder definiert sind, können als Host-Variablen eingesetzt werden.

Ebenso ist es möglich **Datenstruktur-Unterfelder** aus einfachen, intern oder extern beschriebenen Datenstrukturen als Host-Variablen zu verwenden, d. h., MyDS.MyFld kann als Host-Variable verwendet werden, während MyDS. MySubDS.MyFld nicht erlaubt ist.

Das folgende Beispiel zeigt den Quellcode für die Prozedur „GetArtikelInfo“, an die die Artikel-Nr. (PARARTNR) als Input-Only-Parameter übergeben wird. Die Artikel-Bezeichnung (POUTARTBEZ) und der Preis (POUTPREIS) werden an die rufende Prozedur als Output-Parameter zurückgegeben.

Für die übergebene Artikel-Nr. werden aus dem Artikel-Stamm die Artikel-Bezeichnung und der Preis ermittelt. Der Preis (HOSTPREIS) ist als eigenständiges Feld definiert, während die Artikel-Bezeichnung (ARTBEZ) als Datenstruktur-Unterfeld in der qualifizierten Datenstruktur MyHostDS angelegt ist.

```
P GetArtikelInfo  B

D GetArtikelInfo  PI
D   ParArtNr                    22A             Const
D   POutArtBez                  50A    Varying
D   POutPreis                    7P 2

D HostPreis       S              7P 2

D MyHostDS        DS                    Qualified
D   ArtikelBez                  50A    Varying
 *---------------------------------------------------------------
 /Free

     EXEC SQL  Select   ArtBez,      Preis
                  into  :MyHostDS.ArtikelBez, :HostPreis
                  from  ArtikelStX
                  where ArtikelNr = :ParArtNr
               Fetch First Row Only;

     POutArtBez = MyHostDS.ArtikelBez;
     POutPreis  = HostPreis;

     Return;
 /End-Free
P GetArtikelInfo  E
 ******** Quelle EMBHOST1
```

Definition von Host-Variablen

3.2.5.5.2 Host-Strukturen

Bei Host-Strukturen handelt es sich um qualifizierte oder unqualifizierte einfache Datenstrukturen, Mehrfach-Datenstrukturen (Definition mit Schlüssel-Wort OCCURS) oder Array-Datenstrukturen (Definition DIM), in die Ergebnisse aus SELECT-Statements übertragen werden.

Host-Strukturen können auch dazu verwendet werden, um mehrere oder alle Spalten in einem INSERT- oder UPDATE-Statement zu einer Zeile zusammenzufassen.

Bei der Definition von Host-Strukturen sind einige Einschränkungen zu beachten:

- Host-Strukturen können mit Hilfe des Schlüsselworts LIKEDS definiert werden. Verschachtelte Datenstrukturen, d. h. Datenstrukturen, in denen Unterfelder ebenfalls mit dem Schlüsselwort LIKEDS definiert sind, sind jedoch nicht zulässig.
- Host-Strukturen können mit dem Schlüsselwort EXTNAME als externe Datenstrukturen definiert werden. Die Angabe der Feldart (*ALL, *KEY, *INPUT oder *OUTPUT) ist jedoch nicht erlaubt.
- Host-Strukturen können auch mit dem Schlüsselwort LIKEREC definiert werden. Wie auch bei den externen Datenstrukturen wird die Angabe der Feldart (*ALL, *KEY, *INPUT oder *OUTPUT) im zweiten Parameter nicht unterstützt.

Die Datenstruktur-Unterfelder müssen in Anzahl und Definition den im SELECT-, INSERT- oder UPDATE-Befehl aufgelisteten Spalten entsprechen. Wie auch bei der Definition und Verwendung von eigenständigen Host-Variablen können kompatible Datentypen und abweichende Längen verwendet werden. Aus Gründen der Performance und um eine interne Konvertierung zu vermeiden, sollte die Definition der Datenstruktur-Unterfelder mit der Spaltendefinition übereinstimmen.

Das folgende Beispiel zeigt die Funktion „GetUmsatzKunde“, an die die Kunden-Nr. (PARKUNDE) übergeben wird. Für den übergebenen Kunden wird der Gesamtumsatz, die Anzahl der georderten Artikel, sowie der durchschnittliche Umsatz ermittelt Um die Daten zu empfangen, wird eine qualifizierte Datenstruktur DSINFO mit den Unterfeldern Umsatz, Anzahl und Durchschnitt definiert.

```
P GetUmsatzKunde  B

D GetUmsatzKunde  PI
D   ParKundeNr                  15A             Const
D   POutUmsatz                  11P 2
D   POutAnzahl                   5I 0
D   POutDSchnitt                 9S 2

D DSInfo          DS
D   Umsatz                      11P 2
D   Anzahl                       5I 0
D   Durchschnitt                 9S 2
 *---------------------------------------------------------
 /Free
     EXEC SQL  Select   Sum(Wert), Count(Distinct ArtikelNr),
Avg(Wert)
                  into  :DSInfo
                  from  Umsatz
                  where KundeNr = :ParKundeNr
               Fetch First Row Only;

     POutUmsatz    = Umsatz;
     POutAnzahl    = Anzahl;
     POutDSchnitt  = Durchschnitt;

     Return;
 /End-Free
P GetUmsatzKunde  E
 ******* Quelle EMBHOSTDS1
```

Verwendung von Host-Strukturen

Zur Ausgabe der Daten können mehrere Host-Strukturen angegeben werden. Die einzelnen Datenstrukturen müssen durch Kommata getrennt werden.

Das folgende Beispiel zeigt die Prozedur „GetAuftrKopf“. In dieser Prozedur werden für einen als Parameter übergebenen Kunden (PARKUNDENR) alle Spalten aus dem Auftragskopf (Tabelle/Datei AUFTRKOPFX) sowie alle Spalten aus dem zugehörigen Adress-Stamm (Tabelle/Datei ADRESSEX) des ersten Auftrags ermittelt.

Der Auftragskopf wird in eine über das Schlüsselwort LIKEREC definierte Datenstruktur ausgegeben.

3.2.5
Seite 30

Der Adress-Stamm wird in eine externe, über das Schlüsselwort EXTNAME definierte, unqualifizierte Datenstruktur ausgegeben.

```
FAuftrKopfxIF   E             Disk    Rename(Auftrkopf:
   AuftrKopfF)
FAdresseX  IF   E             Disk    Rename(Adresse:
   AdresseF)
 ***************************************************************
P GetAuftrKopf    B

D GetAuftrKopf    PI
D   ParKundeNr                15A     Const
D   POutDSAuftrK                      LikeRec(AuftrKopfF)
D   POutDSAdr                         LikeRec(AdresseF)

D DSAuftrKopf     DS                  LikeRec(AuftrKopfF) Inz
D DSAdresse     E DS                  ExtName(AdresseX)   Inz
D                                     Qualified
 *---------------------------------------------------------------
 /Free

     EXEC SQL Select   *
                 into  :DSAuftrkopf, :DSAdresse
                 from     AuftrKopfx k join Adressex a
                       on k.KundeNr = a.KundeNr
                 Where k.KundeNr = :ParKundeNr
               Fetch First Row Only;

     POutDSAuftrK  = DSAuftrKopf;
     POutDSAdr     = DSAdresse;

     Return;
 /End-Free
P GetAuftrKopf    E
 ****** Quelle EMBHOSTDS2
```

Verwendung mehrerer Host-Strukturen

Vorsicht!

Bei der Verwendung von externen Datenstrukturen muss Folgendes beachtet werden:

Werden Tabellen mit gleichen Schlüsselfeldnamen unter Verwendung der Angabe USING in der JOIN-Anweisung verknüpft, werden seit Release 6.1 nur die Schlüsselfelder der ersten Datei/Tabelle/View ausgegeben, aus der zweiten Tabelle werden diese Spalten eliminiert.

In SQL-Tabellen können Spalten mit dem Schlüsselwort IMPLICITLY HIDDEN definiert werden. Diese Felder werden bei SELECT * nicht ausgegeben, sondern müssen explizit angegeben werden.

Die externen Datenstrukturen (unabhängig davon, ob mit EXTNAME oder LIKEREC definiert) enthalten jedoch ALLE Felder!

Host-Strukturen können auch in INSERT- oder UPDATE-Statements eingesetzt werden, um ganze Zeilen einzufügen oder zu ändern.

Das folgende Beispiel zeigt die Prozedur „UpdAddress“. Diese Prozedur empfängt zunächst die Kunden-Nr. (PARKUNDENR) des Kunden, für den die Adresse geändert werden soll. Im zweiten Parameter (PARDSADRESSE) wird der komplette Datensatz, der die geänderten Werte enthält, übergeben. Diese Datenstruktur wird im UPDATE-Statement angegeben und somit die komplette Adresse (alle Spalten) für den als Parameter übergebenen Kunden aktualisiert. Die Anzahl der geänderten Datensätze (SQLERRD(3) aus der SQLCA wird als Rückgabewert an die rufende Prozedur zurückgegeben.

```
D GblDSAdresse  E DS                  ExtName(AdresseX) Inz
*****************************************************************
P UpdAddress      B

D UpdAddress      PI         10I 0
D   ParKundeNr               15A                          Const
D   ParDSAdresse                    LikeDS(GblDSAdresse) Const
 *---------------------------------------------------------------
 /Free
     EXEC SQL Update AdresseX
                 Set Row = :ParDSAdresse
              Where KundeNr = :ParKundeNr;

     Return SQLERRD(3);
 /End-Free
P UpdAddress      E
 ********* Quelle EMBHOSTDS4
```

Update von Datensätzen mit Host-Strukturen

Mehrfach-Datenstrukturen oder Array-Datenstrukturen können beim multiple-Row-Fetch (Multiple Row Fetches) verwendet werden, um mehrere Datensätze mit einem einzigen SQL-Fetch-Statement auszulesen. Mit Hilfe von Mehrfach- oder Array-Datenstrukturen können auch mehrere Zeilen mit einem einzigen INSERT-Statement eingefügt werden.

Im folgenden Beispiel werden bis zu 20 Aufträge eines vorgegebenen Kunden (HOSTKUNDENR) ermittelt und ausgegeben. Dabei werden die Bestell-Nr. und der Liefertermin der unterschiedlichen Aufträge sowie die Adressinformationen (Name, Strasse, Land, Postleitzahl und Ort) mit einem FETCH-Statement in eine Array-Datenstruktur ausgegeben.

```
D HostKundeNr     S              15A   inz('10002')

D DSAuftraege     DS                   Qualified Dim(20)
D  BestNr                        15A
D  KundeNr                       15A
D    Name                        35A
D    Strasse                     35A
D    Land                         4A
D    PLZ                          9A
D    Ort                         20A
D  LiefTerm                       D   DatFmt(*ISO)

D NbrRows         S               3U 0
 *----------------------------------------------------------
 /Free
   Exec SQL Declare CsrC01 Cursor For
            Select BestNr, k.KundeNr, Name1F, Strasse, Land,
PLZ, Ort,
                   LiefTerm
               from AuftrKopfx k join Adressex a on k.KundeNr
= a.KundeNr
               Where k.KundeNr = :HostKundeNr;

   Exec SQL Open CsrC01;
   Exec SQL Fetch Next From CsrC01 For 20 Rows  into :DSAuf-
traege;
   NbrRows = SQLERRD(3);
   Exec SQL Close CsrC01;

   //Verarbeitung
 /End-Free
 ******* Quelle EMBHOSTDS5
```

Host-Strukturen mit multiplem Row Fetch

3.2.5.5.3 SQL-Datentypen und RPG-Äquivalent

Aus Gründen der Performance bzw. um unnötige Konvertierungen zwischen kompatiblen Datentypen zu vermeiden, sollte man – soweit möglich – die Host-Variablen mit dem gleichen Datentyp und der gleichen Länge wie die Originalspalte definieren.

Es gibt sowohl eine Reihe von RPG-Datentypen, die in SQL nicht unterstützt werden (z. B. Pointer oder Unsigned Integer) als auch SQL-Datentypen, für die es keinen äquivalenten RPG-Datentyp gibt, z. B. Large Objects oder XML.

Werden kompatible Datentypen (z. B. gepackt statt Integer) in der Variablendefinition verwendet, oder werden die Variablen mit abweichender Länge definiert, erfolgt eine interne Konvertierung. Im Falle eines Überlaufs gelten die üblichen RPG-Regeln, d. h. alphanumerische Werte werden abgeschnitten, während beim Überlauf von numerischen oder Datumswerten, ein Fehler ausgegeben wird.

Alphanumerische (Single Byte Character Set – 1 Zeichen = 1 Byte) oder Unicode und Graphic (Double Byte Character Set – 1 Zeichen = 2 Byte) Host-Variablen dürfen nur bis zu der in SQL erlaubten maximalen Spaltenlänge definiert werden.

Für alphanumerische Host-Variablen mit fixer Länge darf die Länge 32.766 Zeichen nicht überschreiten, für alphanumerische Host-Variablen mit variabler Länge liegt die maximale Länge bei 32.740 Zeichen.

Bei Double Byte-Datentypen (C und G) mit fixer Länge sind 16.383 Zeichen erlaubt, während die maximale Länge bei Double Byte-Feldern 16.270 Zeichen nicht überschreiten darf.

Ab Release 6.1 ist es in RPG möglich, alphanumerische (Single und Double Byte) Variablen mit variabler Länge mit dem Schlüsselwort VARYING(2) oder VARYING(4) zu definieren. Der SQL-Precompiler akzeptiert zwar das Schlüsselwort VARYING(2), nicht jedoch die Angabe VARYING(4).

Werden graphische Host-Variablen (Datentyp G) definiert, so wird die in RPG verwendete CCSID (Character Set Id) übernommen.

In SQL-Tabellen oder -Views können Spalten mit Datentypen definiert werden, für die es in RPG kein Äquivalent gibt, z. B. Character Large Objects CLOB oder XML. Dennoch ist es möglich, Werte aus solchen Spalten zu empfangen und in embedded SQL mit Hilfe von SQL-skalaren Funktionen zu bearbeiten. Host-Variablen für SQL-Datentypen, für die es in RPG keine Entsprechung gibt, werden mit Hilfe des Schlüsselworts SQLTYPE und dem SQL-Datentyp definiert.

Die folgende Tabelle gibt einen Überblick über die SQL-Datentypen und deren Übersetzung in RPG.

Datentypen			SQL Datentypen	RPG-Definition	
				Datentypen und Schlüssel-Worte	
alpha-numerisch	Single Byte Character Set (SBCS)	Fixe Länge	CHAR	A	
		Variable Länge	VARCHAR	A	Varying
				A	Varying(2)
		Large Object	CLOB		SQLTYPE(CLOB: Länge)
					SQLTYPE(CLOB_LOCATOR)
					SQLTYPE(CLOB_FILE)
	Double Byte Character Set (DBCS)	Fixe Länge	GRAPHIC	G	CCSID(CharacterSetId)
			GRAPHIC CCSID(13488)	C	
		Variable Länge	VARGRAPHIC	G	Varying CCSID(CharacterSetId)
				G	Varying(2) CCSID(CharacterSetId)
			VARGRAPHIC CCSID(13488)	C	Varying
				C	Varying(2)
		Large Object	DBCLOB		SQLTYPE(DBCLOB: Länge)
					SQLTYPE(DBCLOB_LOCATOR)
					SQLTYPE(DBCLOB_FILE)
	Binär	Fixe Länge	VARBINARY		SQLTYPE(BINARY: Länge)
		Variable Länge	BINARY		SQLTYPE(VARBINARY: Länge)
		Large Object	BLOB		SQLTYPE(BLOB: Länge)
					SQLTYPE(BLOB_LOCATOR)
					SQLTYPE(BLOB_FILE)
numerisch	ganzzahlige Datentypen	Small Integer	SMALLINT	5I 0	
		Integer	INTEGER/INT	10I 0	
		Big Integer	BIGINT	20I 0	
	Feste Anzahl an Dezimal-Positionen	gepackt	DECIMAL/DEC	P	
		gezont	NUMERIC	S	
	Fließkomma-Datentypen	einfache Genauigkeit	REAL	4F	
		doppelte Genauigkeit	DOUBLE	8F	
			DECFLOAT	Nicht unterstützt	
Datum und Zeit		Datum	DATE	D	
		Zeit	TIME	T	
		Zeitmarke	TIMESTAMP	Z	
XML	Single Byte Character Set (SBCS)		XML		SQLTYPE(XML_CLOB: Länge)
					SQLTYPE(XML_CLOB_FILE)
	Double Byte Character Set (DBCS)				SQLTYPE(XML_DBCLOB: Länge)
					SQLTYPE(XML_DBCLOB_FILE)
	Binär				SQLTYPE(XML_BLOB: Länge)
					SQLTYPE(XML_BLOB_FILE)
	unabhängig von Character Set				SQLTYPE(XML_LOCATOR)
ROWID			ROWID		SQLTYPE(ROWID)
DATALINK			DATALINK	Nicht unterstützt	
Result Set Locator			RESULT_SET_LOCATOR		SQLTYPE(RESULT_SET_LOCATOR)

Überblick über SQL-Datentypen und deren RPG-Definition

3.2.5.5.4 Indikator-Variablen

Spalten in SQL-Tabellen können NULL-Werte enthalten. Folglich kann auch das Ergebnis aus SQL-Abfragen NULL-Werte enthalten.

NULL-Werte sind Werte, die außerhalb des gültigen Bereichs des Datentypen liegen. Da RPG-Variablen keine NULL-Werte verarbeiten können, über SQL-Statements jedoch NULL-Werte ausgegeben werden können, gibt es zwei Möglichkeiten, mit NULL-Werten umzugehen:

- Potentielle NULL-Werte werden mit Hilfe von SQL-skalaren Funktionen (COALESCE oder IFNULL) in einen Unterlassungswert konvertiert, der in RPG weiterverarbeitet werden kann.
- NULL-Werte werden über Indikator-Variablen geprüft.

Indikator-Variablen müssen als small Integer (5I 0)-Felder definiert werden.

Indikator-Variablen werden nur durch ein Leerzeichen getrennt von der Host-Variablen, die den zu prüfenden NULL-Wert empfangen kann, angegeben (z. B.:HostVar1 :IndVar1, :HostVar2, :IndVar2). Sofern aus der Abfrage ein NULL-Wert ausgegeben wird, bleibt der Inhalt der Host-Variable unverändert, in die Indikator-Variable wird der Wert -1 ausgegeben. Enthält das Ergebnis keinen NULL-Wert, erfolgt eine Ausgabe in die Host-Variable und die Indikator-Variable erhält den Wert 0.

Im folgenden Beispiel können in allen ausgewählten Spalten NULL-Werte enthalten sein. Für jede Host-Variable wird eine Indikator-Variable definiert und im SELECT … INTO-Statement zugeordnet. Die Indikator-Variablen werden geprüft, um festzustellen, ob in den einzelnen Spalten NULL-Werte ausgegeben wurden.

3.2.5.5.5 Parameter-Marker

Host-Variablen zur Vorgabe von Selektionskriterien zu verwenden, ist nur mit statischem SQL erlaubt, bei dem das SQL-Statement bereits zur Kompilierungszeit bekannt ist. In dynamischem SQL können variable Werte wie folgt gehandhabt werden:

- Die variablen Werte werden **direkt** in den Text **eingebunden**, der das SQL-Statement (Statement-String) enthält.
- Die variablen Werte werden mit **Parameter-Markern** gekennzeichnet.

Parameter-Marker werden in Form von Fragezeichen (?) anstelle der Variablen in den Statement-String eingefügt. Die Host-Variablen, die die eigentlichen Parameterwerte enthalten, müssen entweder beim EXECUTE-Statement (SQL-Statement EXECUTE – sofern ohne Cursor gearbeitet wird) oder beim OPEN-Statement (SQL-Statement OPEN in Verbindung mit Parameter-Markern bei Cursor-Verarbeitung) angegeben werden. Die Reihenfolge der angegebenen Host-Variablen muss der Reihenfolge entsprechen, in der die Parameter-Marker eingebunden werden.

Anmerkung:

Parameter-Marker können nicht dazu verwendet werden, Bibliotheken oder Dateien/Tabellen/Views variabel zu gestalten. Sofern erforderlich, müssen die variablen Werte für Bibliotheken oder Dateien/Tabellen/Views direkt im Statement-String aufgelöst werden.

Im folgenden Beispiel werden für eine beliebige Datei/Tabelle/View, die jedoch die Spalten KUNDENR (Kunden-Nr.) und ARTIKELNR (Artikel-Nr.) enthält, die Anzahl Zeilen für den vorgegebenen Kunden und den vorgegebenen Artikel ermittelt.

Da Dateien/Tabellen/Views nicht über Parameter-Marker gesetzt werden können, wird die Datei/Tabelle/View im Statement-String aufgelöst. Für die Auswahl von Artikel-Nr. und Kunden-Nr. wurden Parameter-Marker verwendet. Die variablen Werte für Kunden-Nr. und Artikel-Nr. werden im OPEN-Statement gesetzt.

Ergänzung 21/2012

3.2.5
Seite 38

```
D Anzahl          S             11P 2
D IndAnzahl       S              5I 0

D String          S            256A    Varying

D ParTable        S             10A    inz('UMSATZ')
D ParArtikelNr    S             22A    inz('ARTNR01')
D ParKundeNr      S             10A    inz('KUNDENR02')
 ***************************************************************
 /Free
    String = 'Select Count(*)             +
                from ' + %Trim(ParTable) +
             ' Where     KundeNr   = ?  +
                       and ArtikelNr = ?  +
             Fetch First Row Only ';

    Exec SQL  Prepare DynSQL from :String;
    ...

    Exec SQL  Declare CsrC01 Cursor For DynSQL;
    ...

    Exec SQL  Open CsrC01 using :ParKundenr, :ParArtikelNr;
    ...

    Clear Anzahl;
    Exec SQL  Fetch Next From CsrC01 into :Anzahl :IndAnzahl;
    Dsply %Char(Anzahl);

    Exec SQL  Close CsrC01;

    *InLR = *On;
 /End-Free
******** Quelle EBMCSRPM1
```

Parameter-Marker zum Setzen von Werten verwenden

Wird VALUES … INTO in dynamischem SQL verwendet, können Parameter-Marker auch für die empfangenden Host-Variablen angegeben werden.

Im folgenden Beispiel wird ebenfalls die Anzahl Zeilen in einer beliebigen Datei/Tabelle/View mit den Spalten KUNDENR (Kunden-Nr.) und ARTIKELNR (Artikel-Nr.) ermittelt. Das Ergebnis wird in die Host-Variable ANZAHL ausgegeben. Das Ausgabefeld wird wie die anderen Variablen als Parameter-Marker gekennzeichnet und im EXECUTE-Statement zugeordnet.

```
D Anzahl          S             11P 2
D IndAnzahl       S              5I 0

D String          S            256A    Varying
D ParTable        S             10A    inz('UMSATZ')
D ParArtikelNr    S             22A    inz('ARTNR01')
D ParKundeNr      S             10A    inz('KUNDENR02')
 ***************************************************************
 /Free
    clear Anzahl;
    String = 'Values(Select Count(*)             +
                        from ' + %Trim(ParTable) +
                      ' where     KundeNr   = ?  +
                                and ArtikelNr = ?  +
                    ) Into ?';

    Exec SQL  Prepare DynSQL01 from :String;
    ...

    Exec SQL  Execute DynSQL01 using :ParKundeNr,
                                     :ParArtikelNr,
                                     :Anzahl :IndAnzahl;
    Dsply SQLCOD;
    Dsply %Char(Anzahl);

    *InLR = *On;
 /End-Free
********** Quelle EMBVALPM1
```

Verwendung von Parameter-Marker für Ausgabe (Quelle DynValPM1)

3.2.5
Seite 40

```
D GePackt         S              11P 2
D Datum           S                D
D Alpha           S              20A

D Gepackt_Ind     S               5I 0
D Datum_Ind       S               5I 0
D Alpha_Ind       S               5I 0
 *---------------------------------------------------------
 /Free
   Exec SQL  Select MyDec, MyDate, MyChar
                into :Gepackt :Gepackt_Ind, :Datum
                     :Datum_Ind, :Alpha :Alpha_Ind
                 From NULLFile
                 Order By MyInt Desc
              Fetch First Row Only;

    If Gepackt_Ind = -1;          //NULL-Wert in Spalte MyDec
       //Verarbeiten NULL-Wert
    EndIf;

    If Datum_Ind = -1;           //NULL-Wert in Spalte MyDate
       //Verarbeiten NULL_Wert
    EndIf;

    If Alpha_Ind = -1;           //NULL-Wert in Spalte MyChar
       //Verarbeiten NULL_Wert
    EndIf;
 /End-Free
 ********* Quelle EMBHOSTIN1
```

Verwendung von Indikator-Variablen

Bei Ausgabe in Host-Strukturen muss für die Indikator-Variablen eine Feldgruppe mit je einem Element pro Unterfeld der Host-Struktur angelegt werden. In das erste Element wird ausgegeben, ob die erste Spalte einen NULL-Wert enthält, in das zweite Element wird die Informationen für die zweite Spalte ausgegeben etc.

Im nächsten Beispiel werden die Werte aus den Spalten MYDEC, MYDATE und MYCHAR in die Datenstruktur DSINFO ausgegeben. Da alle Spalten NULL-Werte enthalten können, müssen sie über Indikator-Variablen geprüft werden. Zum Empfang der Indikator-Variablen wird die Feldgruppe ARRIND definiert und im SELECT … INTO Statement unmittelbar nach der Host-Struktur angegeben. Der Inhalt der Feldgruppenelemente wird geprüft, um festzustellen, ob NULL-Werte ausgegeben wurden.

```
D DSInfo          DS                      Qualified
D   GePackt                        11P 2
D   Datum                           D
D   Alpha                         20A

D ArrInd          S                5I 0   Dim(3)
 *----------------------------------------------------------
 /Free
   Exec SQL  Select MyDec, MyDate, MyChar
                into :DSInfo :ArrInd
                From NULLFile
                Order By MyInt Desc
             Fetch First Row Only;

   If ArrInd(1) = -1;             //NULL-Wert in Spalte MyDec
      //Verarbeiten NULL-Wert;
   EndIf;

   If ArrInd(2) = -1;             //NULL-Wert in Spalte MyDate
      //Verarbeiten NULL-Wert
   EndIf;

   If ArrInd(3) = -1;             //NULL-Wert in Spalte MyChar
      //Verarbeiten NULL-Wert
   EndIf;
 /End-Free
 ********* Quelle EMBHOSTIN2
```

Host-Strukturen mit Indikator-Variablen verwenden

3.2.5.5.6 Namenskonventionen

In RPG gibt es normalerweise keine reservierten Worte, d. h. Variablen-Namen können beliebig festgelegt werden, solange nur die zulässigen Zeichen verwendet werden. So ist es durchaus möglich, Variablen mit dem Namen von Operation Code anzulegen, z. B. UPDATE oder DELETE.

Für die Namensgebung von Host-Variablen sollten jedoch die folgenden Regeln berücksichtigt werden:

- Host-Variablen (und auch andere Variablen, die innerhalb einer Quelle mit embedded SQL verwendet werden), sollten **niemals** mit **SQL, SQ, RDI** oder **DSN** beginnen.

  Diese Präfixe behält sich IBM für zukünftige Erweiterungen im SQL-Precompiler bzw. in der SQL Communications Area und SQL Descriptor Area (SQLDA – SQL Descriptor Area) vor.

  Beispiel:

  Selbst wenn der Variablen-Name SQLCMD heute von IBM nicht verwendet wird, könnte in einem zukünftigen Release eine solche Variable in die SQLCA eingefügt werden. Im besten Fall könnte eine solche Erweiterung einen Kompilierungsfehler verursachen und eine Quellcode-Anpassung erfordern. Schlimmer wäre allerdings, wenn die Programme problemlos erstellt werden könnten und stattdessen unerklärliche Programmfehler auftreten würden.

- Obwohl in RPG Variablen-Namen bis zu 4.096 Zeichen lang definiert werden können, sind Host-Variablen-Namen auf 64 Zeichen beschränkt.

3.2.5.6 Fehlerbehandlung in embedded SQL

Tritt bei Ausführung von embedded SQL-Befehlen ein Fehler auf, so wird im Gegensatz zur Verarbeitung mit native I/O die Verarbeitung hier nicht abgebrochen. Stattdessen wird in der SQLCA, die mit jedem ausgeführten SQL-Statement aktualisiert wird, ein entsprechender Fehlercode ausgegeben.

In der SQLCA werden die Fehlerinformationen in zwei unterschiedlichen Unterfeldern ausgegeben:

- **SQLCOD / SQLCODE**
  Die Ausgabe eines Verarbeitungscodes in diesem Datenstruktur-Unterfeld ist die ursprüngliche AS/400 Art, die jedoch nicht der im SQL-Standard vorgeschriebenen Methode entspricht.

  Der SQLCODE wird als 2-Byte-Binärcode (= 5I 0) ausgegeben.

- **SQLSTT / SQLSTATE**
  Die Ausgabe eines Verarbeitungscodes in diesem Datenstruktur-Unterfeld entspricht der im SQL-Standard vorgeschriebenen Methode.

  Der SQLSTATE wird als fünfstelliger alphanumerischer Wert ausgegeben, der in eine zweistellige Klasse und eine dreistellige Unterklasse aufgeteilt wird.

Anmerkung:

Nach jedem SQL-Statement sollte entweder der SQLCODE oder der SQLSTATE dahingehend geprüft werden, ob die Verarbeitung erfolgreich war.Im Fehlerfall sollten abhängig vom ausgegebenen SQLCODE oder SQLSTATE entsprechende Aktionen eingeleitet werden.

Die folgende Tabelle gibt einen groben Überblick über die unterschiedlichen Funktionsweisen von SQLCODE und SQLSTATE.

	SQLCOD/SQLCODE	SQLSTT/SQLSTATE
Fehler	Negative Zahl	Stelle 1 – 2 ungleich '01', '02', '00'
Nicht gefunden	100	'02000'
Warnung	Positive Zahl außer 100	Stelle 1 – 2 gleich '01'

SQLCODE versus SQLSTATE

Zwischen SQLCODE und SQLSTATE besteht eine n:m-Beziehung, d. h., für einen SQLCODE kann es mehrere SQLSTATEs geben und ein SQLSTATE kann mehreren SQLCODEs zugeordnet werden.

3.2.5.6.1 Fehlerbehandlung über SQLCODE

Die SQLCODEs lassen sich in die folgenden Gruppen unterteilen:

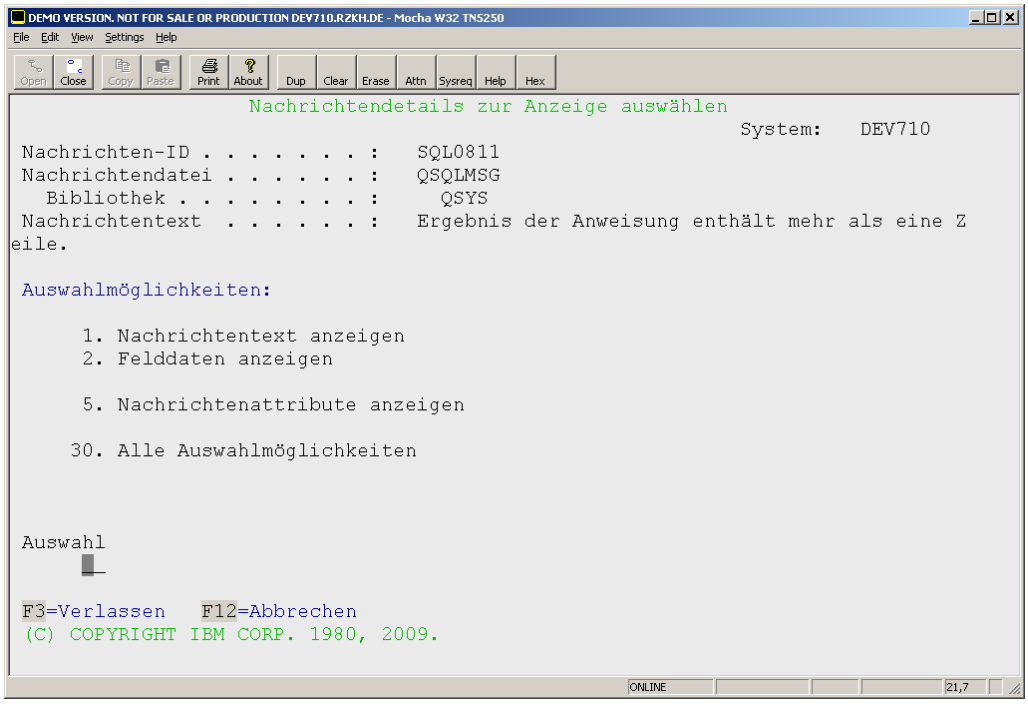
- **SQLCODE = 0**
  Das SQL-Statement wurde ordnungsgemäß verarbeitet.
- **SQLCODE = 100**
  Es wurde entweder kein entsprechender Datensatz gefunden oder bei einer Cursor-Verarbeitung wurde der letzte Datensatz erreicht.
- **SQLCODE < 100**
  Bei der Verarbeitung trat ein Fehler auf.
- **SQLCODE > 0 und nicht 100**
  Es wird eine Warnung ausgegeben.

Die Fehlermeldung, die sich hinter einem SQLCODE verbirgt, lässt sich relativ leicht ermitteln:

- Alle von SQL-Statements ausgegebenen Fehlernachrichten sind in der Nachrichtendatei **QSQLMSG** hinterlegt.
- Die Message-Id der Nachricht setzt sich aus **SQL** (oder **SQ**) und dem absoluten Wert des **SQLCODE**s zusammen, z. B. SQLCODE -811 ergibt die Message-Id SQL0811.

Mit dem CL-Befehl **DSPMSGD** (Nachrichtenbeschreibung anzeigen) kann der Fehlertext einfach angezeigt werden, wie das folgende Beispiel zeigt:

```
DSPMSGD RANGE(SQL0811)
MSGF(QSQLMSG)
```



Anzeige SQL-Fehler mit CL-Befehl DSPMSGD (Nachrichtenbeschreibung anzeigen)

Mit Hilfe des CL-Befehls RTVMSG (Nachricht auffinden) kann die Fehlermeldung auch über ein kleines CL-Programm ermittelt werden. Variable Nachrichtentexte werden ebenfalls in der SQLCA, im Datenstruktur-Unterfeld SQLERRMC/SQLERM, ausgegeben und können im RTVMSG-Befehl in der Option MSGDTA eingesetzt werden.

```
RTVMSG MSGID(&MSGID)
MSGF(QSQLMSG)
MSGDTA(&SQLERM)
MSG(&MSGTEXT)
```

SQL-Nachrichtentext mit CL-Befehl RTVMSG (Nachricht auffinden) ermitteln

Seit Release V5R3M0 kann der SQL-Message-Text auch über die Diagnostics Area, d. h. den SQL-Befehl GET DIAGNOSTICS ermittelt werden.

3.2.5.6.2 Fehlerbehandlung über SQLSTATE

Im SQL-Standard wird die ordnungsgemäße Verarbeitung bzw. eine Fehlersituation über den SQL-Status gesteuert. Bei dem SQL-Status handelt es sich um einen fünfstelligen alphanumerischen Wert.

Dieser fünfstellige alphanumerische Status kann in eine Statusklasse (Stelle 1-2) und Unter-/Subklasse (Stelle 3-5) aufgeteilt werden. Über die Klasse kann festgestellt werden, ob ein SQL-Statement erfolgreich ausgeführt, eine Warnung ausgegeben oder kein Satz gefunden wurde:

- **Klasse 00**
  Das SQL-Statement wurde erfolgreich ausgeführt.
- **Klasse 01**
  Es wurde eine Warnung ausgegeben.
- **Klasse 02**
  Es wurde kein Datensatz gefunden.
- **Alle anderen Klassen**
  Es wurde ein Fehler ausgegeben.

  Die Klassen, die mit den Ziffern 0-6 und den Buchstaben A-H beginnen, sind für den Datenbanken-Manager reserviert. Innerhalb dieser Klassen sind wiederum die Unterklassen, die mit den Ziffern 0-9 und den Buchstaben A-H beginnen, ebenfalls für den Datenbanken-Manager reserviert.

  SQL-Stati außerhalb dieser reservierten Klassen und Subklassen können in der SQL-Programmierung individuell belegt und geprüft werden.

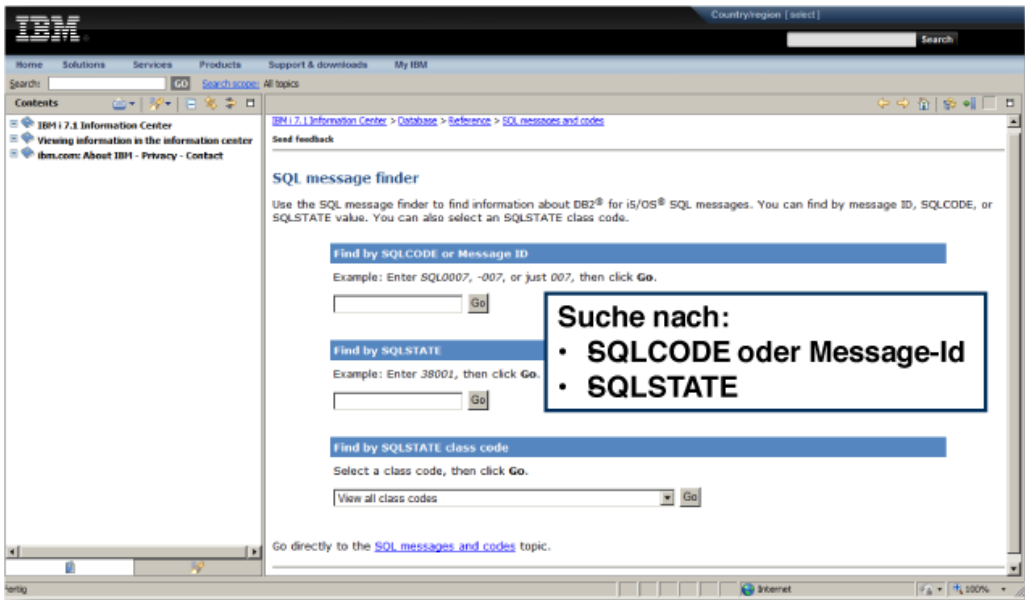
3.2.5.6.3 IBM Information Center – SQLSTATEs und SQLCODEs

Während die Fehlernachrichten für SQLCODEs einfach aus der Message-File QSQLMSG ermittelt werden können, steht für die SQLSTATEs kein vergleichbarer Mechanismus zur Verfügung.

Im IBM Information Center sind die gewünschten Informationen jedoch unter diversen Links verfügbar:

- **SQL-Status-Klassen:**
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/topic/rzala/rzalaclass.htm
- **SQL-Stati:**
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/topic/rzala/rzalaccl.htm
- **SQL-Nachrichten:**
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/topic/rzala/rzalaml.htm
- **SQL-Nachrichten-Finder:**
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/topic/rzas2/rzas2finder.htm

Die folgende Abbildung zeigt den SQL-Message-Finder im IBM Information Center:



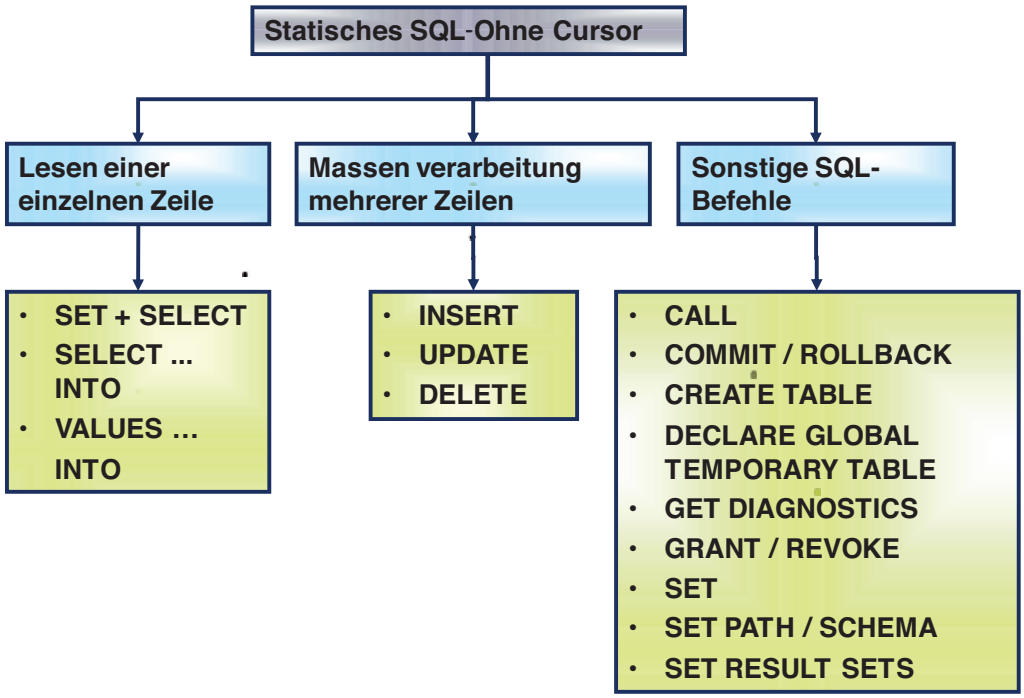
IBM Information Center – SQL Message Finder

3.2.5.7 Statisches SQL ohne Cursor

Bei statischem SQL ist das SQL-Statement bereits zur Laufzeit bekannt. Durch die Verwendung von Host-Variablen können diese Statements zur Laufzeit für unterschiedliche Werte ausgeführt werden.

Sofern gelesene Datensätze/Zeilen nicht einzeln verarbeitet werden müssen, ist eine Cursor-Verarbeitung nicht erforderlich.

Das folgende Schaubild gibt einen Überblick über die wichtigsten SQL-Befehle, die mit statischem SQL ohne Cursor verarbeitet werden können.



Überblick über SQL-Befehle ohne Cursor-Verarbeitung

3.2.5.7.1 SQL-Befehl SET

Bei dem SQL-Befehl SET handelt es sich um den Zuordnungsbefehl, der in der SQL-Programmierung ähnlich wie in RPG der OpCode EVAL eingesetzt wird.

Mit Hilfe des SQL-Statements SET können SQL Special Registers und SQL-skalare Funktionen mit embedded SQL in RPG oder COBOL verwendet werden, ohne dass ein Zugriff auf eine Datei/Tabelle/View erforderlich ist.

Setzen von einzelnen Host-Variablen
Mit dem SQL-Statement SET können **einzelne Host-Variable** gesetzt werden.

Bei dem Ausdruck kann es sich z. B. um ein Special Register, eine skalare Funktion, eine Berechnung (von numerischen oder Datums- und Zeit-Werten) oder eine Verknüpfung von Textinformationen handeln. Innerhalb des Ausdrucks können weitere Host-Variablen verwendet werden.

```
/Free
//Syntax
Exec SQL Set :HostVariable = Ausdruck;
```

Syntax SET-Statement – Setzen von einzelnen Variablen

Im folgenden Beispiel wird zunächst die Variable MYNEWDATE über das Special Register CURRENT_DATE mit dem Tagesdatum versorgt. Im zweiten SET-Statement werden alle Buchstaben in der Host-Variable MYTEXT mit Hilfe der skalaren Funktion UPPER in Großbuchstaben konvertiert.

```
D MyText          S             50A   Varying inz('Text
   in großen und +
D                                      kleinen Buchstaben')
D MyNewDate       S               Z
 *----------------------------------------------------
 /Free
   Exec SQL  Set :MyNewDate = Current_Timestamp;

   Exec SQL  Set :MyText = Upper(:MyText);
   ...
 /End-Free
```

SET-Statement – Setzen von einzelnen Host-Variablen – Beispiel

Setzen von mehreren Host-Variablen
In einem einzigen SQL SET-Statement können gleichzeitig **mehrere Host-Variablen** gesetzt werden.

Auf der linken Seite des Gleichheitszeichens werden die zu versorgenden H**ost-Variablen,** durch Kommata getrennt, innerhalb einer **runden Klammer** aufgelistet. Auf der rechten Seite des Gleichheitszeichens werden die einzelnen Ausdrücke ebenfalls durch Kommata getrennt und in eine **runde Klammer** eingebettet angegeben.

```
 /Free
   //Syntax
     Exec SQL  Set (:HostVar1, :HostVar2, ... :HostVarN) =
                    (Ausdruck1, Ausdruck2, ... AusdruckN);
```

Syntax SET-Statement – Setzen von mehreren Variablen

Im folgenden Beispiel werden die Host-Variablen NAME, STRASSE und PLZORT geändert. Beim Namen werden alle Buchstaben in Großschrift konvertiert, bei der Straße wird alles in Kleinbuchstaben umgesetzt. Die Variable PLZORT wird aus den Variablen PLZ und ORT gebildet. Von beiden Variablen werden die führenden und folgenden Blanks entfernt. Der Ort wird zudem in Großbuchstaben konvertiert. Postleitzahl und Ort werden durch einen Bindestrich getrennt.

```
/Free
   ...
  Exec SQL  Set (:Name, :Straße, :PLZOrt) =
               (Upper(:Name), Lower(:Straße),
               Trim(:PLZ) concat ' - ' Concat Upper(Trim
                 (:Ort)));
   ...
/End-Free
 ************ Quelle EMBSET01
```

SET-Statement – Setzen von mehreren Host-Variablen – Beispiel

Setzen von mehreren Host-Variablen über SELECT-Statement
Mit einem SQL-Set-Statement können eine oder **mehrere Host-Variablen** über ein SQL-**SELECT-Statement** gefüllt werden.

Voraussetzung ist, dass das SQL-SELECT-Statement **genau einen** Datensatz ausgibt.

Host-Variablen und SELECT-Statement müssen mit **runden Klammern** umschlossen werden, auch dann, wenn nur eine einzige Host-Variable gefüllt wird.

```
/Free
   //Syntax
    Exec SQL  Set (:HostVar1, :HostVar2, ... :HostVarN) =
                  (Select  Col1, Col2, ... ColN
                      From ...);
```

Syntax SET-Statement mit SELECT-Befehl

Im folgenden Beispiel werden die Summe (SUMME), das Maximum (MAXIMUM) und die Anzahl von Positionen (ANZAHL) für einen vorgegebenen Kunden (PARKUNDENR) innerhalb eines vorgegebenen Zeitraums (VONDATUM / BISDATUM) aus der Umsatztabelle (UMSATZ) ermittelt.

```
/Free
  //Beispiel:
    Exec SQL  Set (:Summe, :Maximum, :Anzahl) =
                  (Select  Sum(Wert), Max(Wert), Count(*)
                      From Umsatz
                      Where     KundeNr = :ParKundeNr
                            And VerkDatum between :VonDatum
                                              And :BisDatum);
   ...
/End-Free
******** Quelle EMBSET02
```

SET-Statement in Verbindung mit SELECT-Befehl – Beispiel

3.2.5.7.1.1 Verwendung von SPECIAL REGISTERs

SQL bietet über Special Registers eine Reihe von Informationen, die in RPG ansonsten entweder nicht oder nur eingeschränkt zur Verfügung stehen.

So gibt es z. B. das Special Register CURRENT_TIMEZONE, das die Abweichung zur UTC (Universal Time Coordinated / Greenwich Time) in Stunden angibt. Damit kann die aktuelle Zeit in unterschiedlichen Zeitzonen oder die UTC einfach ermittelt werden.

Das Special Register CURRENT_TIMESTAMP liefert die aktuelle Zeitmarke. In RPG kann die aktuelle Zeitmarke zwar auch über die Built-In-Function %TIMESTAMP() ermittelt werden, das SQL Special Register liefert jedoch genauere Werte. Die RPG-Built-In-Function versorgt nur die ersten drei Stellen der Mikrosekunden, während bei dem SQL Special Register alle sechs Stellen ausgegeben werden.

Im folgenden Beispiel wird zunächst der Benutzer über das Special Register SESSION_USER ermittelt. Im zweiten SQL-Statement werden sowohl die aktuelle Zeitmarke als auch die aktuelle Zeitzone über die Special Registers CURRENT_TIMESTAMP und CURRENT_TIMEZONE ermittelt. Im letzten SQL-Statement wird die UTC basierend auf den Special Registers CURRENT_TIME und CURRENT_TIMESZONE berechnet und ausgegeben.

```
D Benutzer        S             10A

D ZeitMarke       S               Z
D ZeitZone        S              6P 0
D GreenwichZeit   S               T
*---------------------------------------------------------
 /Free
       EXEC SQL  Set :Benutzer = Session_User;
       Dsply Benutzer;

       Exec SQL  Set (:Zeitmarke, :ZeitZone) =
                     (Current_Timestamp, Current_Timezone);
       Dsply (%Char(ZeitMarke) + ' ' + %Char(ZeitZone));

       EXEC SQL  Set :GreenwichZeit = Current_Time -
         Current_Timezone;
       Dsply %Char(GreenwichZeit);
    ...
 /End-Free
 ************ Quelle EMBSPCREG1
```

Verwendung von SQL Special Registers in embedded SQL

3.2.5.7.1.2 Berechnung von Zeitdifferenzen

In RPG können Datums- und Zeitdifferenzen über den OpCode SUBDUR oder die Built-In-Function %DIFF() ermittelt werden. Die Differenz wird dabei immer in der vorgegebenen Zeiteinheit, z. B. Tage oder Stunden ermittelt.

Benötigt man die Differenz zwischen zwei Zeiten in Stunden, Minuten, Sekunden, muss zunächst die Differenz in Sekunden ermittelt werden. Dieser Betrag muss durch 3600 dividiert werden, um die Anzahl der Stunden zu berechnen. Der Rest muss durch 60 dividiert werden, um die Anzahl der Minuten zu ermitteln. Bei dem Restwert handelt es sich um die Anzahl Sekunden. Die Berechnung wird im folgenden Beispiel gezeigt.

```
D Zeit1           S               T   inz(T'18.30.45')
D Zeit2           S               T   inz(T'10.00.00')

D DiffZeitSek     S              9P 0
D DSDiffZeit      DS
D   DiffZeit                     6S 0
D   Stunden                      2S 0  Overlay(DiffZeit)
D   Minuten                      2S 0  Overlay(DiffZeit:
                                         *Next)
D   Sekunden                     2S 0  Overlay(DiffZeit:
                                         *Next)
 *----------------------------------------------------------
 /Free
    DiffZeitSek = %Diff(Zeit1: Zeit2: *Seconds);
    Stunden     = %Div(DiffZeitSek: 3600);
    Minuten     = %Div(%Rem(DiffZeitSek: 3600): 60);
    Sekunden    = %Rem(%Rem(DiffZeitSek: 3600): 60);
    Dsply DiffZeit;
    ...
 /End-Free
 ********* Quelle EMBTIME1
```

Berechnung von Zeitdifferenzen in Stunden, Minuten, Sekunden mit RPG

Wird eine Zeitdifferenz mit SQL ermittelt, so wird die Differenz per Default nicht in einer vorgegebenen Zeiteinheit (z. B.Tage oder Minuten) ausgegeben, sondern als Kombination von Stunden, Minuten, Sekunden (Zeit), Jahren, Monaten, Tagen. (Datum) bzw. in Jahren, Monaten, Tagen, Stunden, Minuten, Sekunden und Mikrosekunden (Zeitmarke).

Sofern die Zeitdifferenz mit SQL in einer bestimmten Zeiteinheit ermittelt werden soll, müssen in SQL skalare Funktionen wie DAYS() oder MIDNIGHT_SECONDS() oder TIMESTAMP_DIFF() verwendet werden.

Werden zwei Zeiten mit (embedded) SQL voneinander abgezogen, so wird das Ergebnis als sechsstelliger numerischer Wert ausgegeben. In den ersten beiden Stellen des Ergebnisses wird die Differenz in Stunden, in den nächsten beiden Stellen werden die Minuten und in den letzten beiden Stellen die Sekunden ausgegeben. Das Ergebnis aus der Subtraktion der Zeiten 14:30:15 und 10:00:00 ergibt demnach 43015 (= 4 Stunden, 30 Minuten und 15 Sekunden).

Die Differenz zwischen zwei Datumsfeldern wird als achtstelliger numerischer Wert im Format JJJJMMTT ausgegeben. Die Differenz zwischen zwei Zeitmarken wird als 20-stelliger numerischer Wert mit sechs Dezimal-Positionen im Format JJJJMMDDHHMMSS,MSMSMS ausgegeben.

Im folgenden Beispiel wird die Differenz zwischen den gleichen Zeitwerten wie im vorherigen Beispiel mit embedded SQL berechnet.

```
D Zeit1           S               T   inz(T'18.30.45')
D Zeit2           S               T   inz(T'10.00.00')

D DSDiffZeit      DS
D   DiffZeit                     6S 0
D   Stunden                      2S 0  Overlay(DiffZeit)
D   Minuten                      2S 0  Overlay(DiffZeit:
                                         *Next)
D   Sekunden                     2S 0  Overlay(DiffZeit:
                                         *Next)
*----------------------------------------------------------
/Free
    Exec SQL  Set :DiffZeit = :Zeit1 - :Zeit2;
    Dsply DiffZeit;
    ...
/End-Free
******* Quelle EMBTIME1
```

Berechnung von Zeitdifferenzen mit embedded SQL

3.2.5.7.1.3 Verwendung von SQL-Prädikaten

SQL beinhaltet einige Prädikate, für die es in RPG kein Äquivalent gibt, z. B. LIKE oder IN. Anstatt durch aufwendigen RPG- oder Cobol-Quellcode diese Funktionalität nachzubilden, können SQL-Prädikate einfach mit embedded SQL verwendet werden.

Das folgende Beispiel zeigt die Funktion LIKE, an die sowohl der zu durchsuchende Text als auch der Text (inklusive Platzhalter % für beliebig viele Zeichen und/oder _ für einzelne Zeichen), nach dem gesucht werden soll, übergeben werden. Sofern es erforderlich ist, nach den Platzhaltern % oder _ zu suchen, kann ein ESCAPE-Zeichen übergeben werden, das im Vergleichstext unmittelbar vor dem Sonderzeichen angegeben werden muss.

Beispiel: Sofern nach „Skonto“ und folgendem „%“ gesucht werden soll, kann der Text nach dem gesucht werden soll, wie folgt angegeben werden: ‘Skonto%!%%‘.

Das Ausrufezeichen wird als ESCAPE-Zeichen verwendet.

Sofern der zu suchende String in dem übergebenen Text (mit oder ohne ESCAPE-Zeichen) gefunden wird, wird *ON an die rufende Prozedur ausgegeben, anderenfalls *OFF. Durch die Sortierreihenfolge (SRTSEQ) *LangIdShr, die in der SET OPTION-Anweisung gesetzt wird, erfolgt die Suche unabhängig von Groß- oder Kleinschreibung (case insensitive).

```
D IsFound         S               N
 *---------------------------------------------------------
C/Exec SQL Set Option Commit=*NONE, DatFmt=*ISO,
SrtSeq=*LangIdShr
C/End-Exec
 ***********************************************************
 /Free
   //Beispiele für den Aufruf
   //1. Suche nach IST und folgendem EINFACH
        (case insensitive)
   IsFound = Like('Alles ist zunächst schwierig,
                  bevor es einfach wird':
                  '%IST%EINFACH%');
   Dsply IsFound;

   //2. Suche nach 4 % (ESCAPE-Zeichen ist die Pipe |)
   Clear IsFound;
```

3.2.5
Seite 56

```
   IsFound = Like('Rabatt von 3 %, 4 % oder 7.5 %':
                  '%4 |%%': '|');
   Dsply IsFound;

   *INLR = *On;
 /End-Free
 *************************************************************
 * Procedure Name:   Like
 * Purpose:          Wrapper für SQL Prädikat Like
 *                   (case insensitive Suche)
 * Returns:          *ON = gefunden / *OFF = nicht gefunden
 * Parameter:        ParText   => Quelltext, der durchsucht
 *                                werden soll
 * Parameter:        ParSearch => Text, nach dem gesucht
 *                                werden soll
 * Parameter:        ParEsc    => Escape Zeichen
 *                             => wenn nach % oder _ gesucht
 *                                werden soll
 *************************************************************
P Like            B

D Like            PI              N
D  ParText                      256A   Varying Const
D  ParSearch                     50A   Varying Const
D  ParEsc                         1A   Const Options(*NoPass)

D Found           S               N
 *-------------------------------------------------------------
 /Free
   If %Parms >= %ParmNum(ParEsc);
      Exec SQL Set :Found = Case When Trim(:ParText) like
Trim(:ParSearch)
                                    Escape :ParEsc
                                Then '1' else '0' End;
   Else;
      Exec SQL Set :Found = Case When Trim(:ParText) like
Trim(:ParSearch)
                                Then '1' else '0' End;
   EndIf;
```

```
    Return Found;

  /End-Free
 P Like            E
  ************* Quelle EMBLIKE
```

Verwendung von SQL-Prädikaten in embedded SQL

3.2.5.7.1.4 Verwendung von SQL-skalaren Funktionen

SQL liefert eine Reihe von skalaren Funktionen, für die es in RPG entweder kein Äquivalent gibt, oder die in RPG eine andere Funktionalität aufweisen.

SQL liefert z. B. skalare Funktionen, mit deren Hilfe beliebige Texte und Groß- oder Kleinbuchstaben konvertiert werden können, ohne dass auf sprachenindividuelle Sonderzeichen Rücksicht genommen werden müsste.

Rundungsfunktionen erlauben, einen numerischen Wert auf die nächste Ganzzahl auf- oder abzurunden. Beim kaufmännischen Runden kann nicht nur auf die Anzahl der Dezimalstellen, sondern auch auf den vollen Zehner, Hunderter oder Tausender gerundet werden.

Im Bereich der Datumsrechnung wird eine Reihe von skalaren Funktionen zur Verfügung gestellt, über die u. a. der Wochentag, der Name des Wochentags oder Monats, die Kalenderwoche nach ISO-Richtlinien, das Monatsende oder das Datum z. B. des nächsten Mittwochs ermittelt werden kann.

Weiterhin liefert SQL trigonometrische Funktionen – z. B. SIN(), COS() – und Funktionen zur Berechnung von Logarithmen, z. B. LN10(), LOG10().

Die folgende Tabelle gibt einen Überblick über die wichtigsten SQL-skalaren Funktionen, für die es in RPG keine Entsprechung gibt, die jedoch mit embedded SQL einfach genutzt werden können.

Gruppe	Funktion	Beschreibung	Gruppe	Funktion	Beschreibung
Datums- und Zeitrechnung	DAYNAME	Name des Wochentags	Runden	CEILING	Aufrunden auf die nächste ganze Zahl
	DAYOFWEEK	Numer. Wochentag (Sonntag = 1)		FLOOR	Abrunden auf die nächste ganze Zahl
	DAYOFWEEK_ISO	Numer. Wochentag (Montag = 1)		ROUND	Kaufmännisch Runden (ind. Rundung auf 10er, 100er usw.)
	DAYOFYEAR	lfd. Tag im Jahr	Funktionen zum Ver- und Entschlüsseln von Daten	ENCRYPT_RC2	Verschlüsseln nach RC2 (Rivest Cipher)
	MONTHNAME	Monatsname		ENCRYPT_TDES	Verschlüsseln nach Triple DES (Data Encryption Standard)
	QUARTER	Quartal		ENCRYPT_AES	Verschlüsseln nach AES (Advanced Encryption Standard)
	NEXT_DAY	Datum des nächsten Montags, Dienstags …		GETHINT	Ermitteln Hilfestellung für Encryption Passwort
	LAST_DAY	Monatsende		DECRYPT_BIT	Entschlüsseln nach Bit-Format
	ADD_MONTHS	Monate addieren: Wenn Ausgangsdatum = Monatsende dann Zieldatum = Monatsende		DECRYPT_BINARY	Entschlüsseln nach Binär-Format
	MONTHS_BETWEEN	Differenz in Monaten und Dezimal-Tagen		DECRYPT_CHAR	Entschlüsseln nach Single-Byte/alphanumerisches Format
	WEEK	Woche (1. Januar immer in KW 1)		DECRYPT_DB	Entschlüsseln nach Double Byte-Format
	WEEK_ISO	Kalenderwoche nach ISO (Mehrheit der Tage im neuen Jahr in KW 1, Montag = 1. Wochentag)	String-Funktionen	UPPER	Konvertieren in Großschrift
	TIMESTAMP_FORMAT	Konvertieren String in Zeitmarke		LOWER	Konvertieren in Kleinschrift
	VARCHAR_FORMAT	Konvertieren Zeitmarke in String		REPEAT	Zeichen wiederholen
	ROUND_TIMESTAMP	Auf Zeiteinheit runden		REPLACE	Ersetzen/Entfernen alle Vorkommen
	TRUNC_TIMESTAMP	Auf Zeiteinheit abschneiden		SPACE	Wiederholen Anzahl Blanks
	GENERATE_UNIQUE	Ermitteln eindeutigen Wert auf Basis der aktuellen Zeitmarke		LEFT	Extrahieren Anzahl Zeichen von der linken Seite aus
				RIGHT	Extrahieren Anzahl Zeichen von der rechten Seite aus

SQL-skalare Funktionen ohne Entsprechung in RPG

Mit Hilfe des SET-Statements können SQL-skalare Funktionen direkt ohne Dateizugriff in RPG verwendet werden.

Das folgende Beispiel zeigt die Prozedur YearWeek, an die ein Datum übergeben wird.

Für dieses Datum wird die Kalenderwoche nach ISO-Richtlinien ermittelt (= Woche beginnt mit Montag und in der ersten Woche des Jahres muss die Mehrzahl der Tage aus dem neuen Jahr sein) sowie das Jahr, in dem die Kalenderwoche liegt.

Die Kalenderwoche wird über die skalare Funktion WEEK_ISO ermittelt.

Sofern die Kalenderwoche größer oder gleich 51 ist und der Monat des übergebenen Datums der Januar ist, wird das Vorjahr ausgegeben. Sofern die erste Kalenderwoche ermittelt wurde und der Monat des übergebenen Datums der Dezember ist, wird das Folgejahr ausgegeben. Ansonsten wird das Jahr des übergebenen Datums ausgegeben. Das ermittelte Jahr wird in einen alphanumerischen Wert konvertiert und mit der ebenfalls in einen alphanumerischen Wert konvertierten Kalenderwoche verbunden.

```
*************************************************************
 * Procedure Name:   YearWeek
 * Purpose:          Ermitteln Jahr/Kalenderwoche nach ISO-
                       Richtlinien
 * Returns:          Jahr/KW
 * Parameter:        ParDate       => Datum
 *************************************************************
P YearWeek        B

D YearWeek        PI             7A
D  ParDate                        D   DatFmt(*ISO) Const

D RtnWeek         S              7A
D NumWeek         S              2P 0
 *-------------------------------------------------------------
 /Free
   Exec SQL  Set :NumWeek = Week_ISO(:ParDate);

   Exec SQL
    Set :RtnWeek = VarChar(Case When :NumWeek > 51 and
Month(:ParDate) =  1
                               Then Year(:ParDate) - 1
                               When :NumWeek =  1 and
Month(:ParDate) = 12
                               Then Year(:ParDate) + 1
                               Else Year(:ParDate) End)
                  concat '/' Concat VarChar(:NumWeek);

   Return RtnWeek;
 /End-Free
P YearWeek        E
 ********** Quelle EMBWEEK1
```

Verwendung von SQL-skalaren Funktionen in embedded SQL

Ergänzung 21/2012

3.2.5.7.2 Massendatenverarbeitung

Mit einem einzigen SQL-Statement können mehre, ja sogar Hunderte oder Tausende von Datensätzen geändert, hinzugefügt oder gelöscht werden.

Hierzu ist es im Gegensatz zum Record Level Access nicht notwendig, jeden Satz einzeln zu lesen und zu verarbeiten. Die Verarbeitung kann en bloc erfolgen.

Durch die Verwendung von Host-Variablen in den WHERE-Bedingungen, die in UPDATE oder DELETE-Statements verwendet werden können, können die Daten eingeschränkt werden.

Im folgenden Beispiel werden alle Aufträge mit Status ERL (= erledigt) in History-Dateien übertragen und aus den Originaldateien gelöscht.

In den History-Dateien sind die gleichen Spalten in der gleichen Reihenfolge wie in den Originaldateien angelegt. Die History-Dateien enthalten zwei zusätzliche Spalten, in denen der Zeitpunkt der Übernahme sowie der Benutzer, der die Übernahme initiiert hat, eingetragen werden.

Zuerst werden die Auftragspositionen in die History-Datei für die Auftragspositionen übernommen. Danach werden die erledigten Auftragsköpfe kopiert.

Anschließend werden zunächst die erledigten Auftragspositionen und zum Schluss die erledigten Auftragsköpfe aus den Originaldateien gelöscht.

Sollte in einem der SQL-Statements ein Fehler auftreten, so werden alle kopierten und gelöschten Datensätze zurückgesetzt (ROLLBACK) und das Programm beendet. Nach erfolgreicher Übernahme werden die Änderungen festgeschrieben (COMMIT).

```
D HostStatus      S              3A     inz('ERL')
 *-------------------------------------------------------------
 /Free
    *INLR = *On;
    EXEC SQL Commit;

    //Übernahme Auftragspositionen in History-Datei
    EXEC SQL Insert Into AuftrPosH
             Select  p.*, Current_Timestamp, Session_User
                From AuftrKopfx k join AuftrPosx p
                     On     k.Firma  = p.Firma
                        And k.BestNr = p.BestNr
```

```
                Where k.Status = :HostStatus;
    If SQLCODE < *Zeros;    //Fehler
       Exec SQL Rollback;
       Return;
    EndIf;

    //Übernahme Auftragsköpfe in History-Datei
    EXEC SQL Insert Into AuftrKopfH
             Select  k.*, Current_Timestamp, Session_User
                From AuftrKopfx k
                Where k.Status = :HostStatus;
    If SQLCODE < *Zeros;     //Fehler
        Exec SQL Rollback;
        Return;
    EndIf;

    //Löschen erledigte Auftragspositionen aus
        Auftragspositionsdatei
    EXEC SQL  Delete From AuftrPosX p
                Where Exists (Select 1 from AuftrKopfX k
                        Where      k.Firma  = p.Firma
                              And  k.BestNr = p.BestNr
                              And  k.Status = :HostStatus);
    If SQLCODE < *Zeros;     //Fehler
        Exec SQL Rollback;
        Return;
    EndIf;

    //Löschen erledigte Auftragsköpfe aus Auftragskopfdatei
    EXEC SQL  Delete From AuftrKopfX
                Where Status = :HostStatus;
    If SQLCODE < *Zeros;    //Fehler
        Exec SQL Rollback;
        Return;
    EndIf;

    Exec SQL Commit;
    Return;
  /End-Free
  ************* Quelle EMBREORG1
```

Massendatenverarbeitung

3.2.5.7.3 Einzelsatz-Verarbeitung

Sofern nur ein einzelner Datensatz eingelesen werden soll, ist noch keine Cursor-Verarbeitung notwendig.

Es gibt drei Möglichkeiten, einen einzelnen Datensatz aus einem SQL SELECT-Statement einzulesen:

- SET-Statement in Verbindung mit einer SELECT-Anweisung (1.6.3.1)
- SELECT … INTO-Statement (1.6.3.2)
- VALUES … INTO (1.6.3.3)

3.2.5.7.3.1 SQL-Befehl SET in Verbindung mit SELECT-Anweisung

Das Ergebnis muss **genau eine Zeile** ergeben. Wenn das Ergebnis mehr als eine Zeile enthält, wird ein Fehler ausgegeben. Im Fehlerfall werden in die Host-Variablen keine Daten ausgegeben.

Host-Variablen und SELECT-Statement müssen durch **runde Klammern** umschlossen werden, auch dann, wenn nur eine einzige Host-Variable gefüllt wird.

```
/Free
  //Syntax
    Exec SQL  Set (:HostVar1, :HostVar2, ... :HostVarN) =
                  (Select  Col1, Col2, ... ColN
                      From ...);
```

Syntax SET-Statement mit SELECT-Anweisung

In folgendem Beispiel wird der kumulierte Umsatz für einen vorgegebenen Kunden (Host-Variable PARKUNDENR) und einen vorgegebenen Zeitraum (Host-Variablen VONDATUM und BISDATUM) ermittelt und in die Host-Variable *Gesamt* ausgegeben.

```
D ParKundeNr      S              15A     inz('10002')
D VonDatum        S                D     inz(D'2011-01-01')
D BisDatum        S                D     inz(*Sys)
D Gesamt          S              11P 2
*-------------------------------------------------------------
/Free
  *INLR = *ON;

  //Ermitteln kumulierten Umsatz für Kunde in Zeitraum
    Exec SQL  Set (:Gesamt) = (Select Sum(Wert)
                        from Umsatz
                        Where     KundeNr = :ParKundeNr
                             and VerkDatum between :VonDatum
                                               and :BisDatum);
    If SQLCODE < *Zeros;
       //Fehlerbehandlung
    EndIf;

    Dsply %Char(Gesamt);
    Return;
 /End-Free
 ********* Quelle EMBSET02
```

Beispiel SET-Statement mit SELECT-Anweisung

3.2.5.7.3.2 SQL-Befehl SELECT ... INTO

Die Syntax entspricht dem SELECT-Statement (inklusive Common Table Expressions). Die empfangenden Host-Variablen oder Host-Strukturen werden zwischen SELECT- und FROM-Anweisung in der INTO-Anweisung angegeben.

Die Anzahl der Host-Variablen (bzw. die Anzahl der Unterfelder in Host-Strukturen) muss der Anzahl der ausgewählten Spalten entsprechen. Die Host-Variablen müssen zumindest in einem kompatiblen Datentypen definiert sein.

Die Angabe von Indikator-Variablen ist optional bzw. nur erforderlich, wenn NULL-Werte ausgegeben werden könnten.

```
/Free
   //Syntax
   Exec SQL  Select Col1, Col2, ... ColN
                  Into :HostVar1 :IndVar1, :HostVar2, ...
:HostVarN :InVarN
               From ... ;
```

Syntax SELECT ... INTO-Statement

Das Ergebnis aus der SELECT-Anweisung sollte entweder keine oder genau eine Zeile ergeben.

Wird keine Zeile ausgegeben, so wird SQLCODE 100 bzw. SQLSTATE 02000 ausgegeben. Die Host-Variablen werden **nicht initialisiert.**

Enthält das Ergebnis mehr als eine Zeile, wird ein Fehler ausgegeben (SQLCODE -811 bzw. SQLSTATE 21000). Im Fehlerfall können Daten in die Host-Variablen ausgegeben werden.

Anmerkung:

Werden die Host-Variablen nicht gefunden, dann können sie nicht initialisiert werden. Wenn bei der Ausgabe von mehreren Datensätzen nicht sicher ist, ob die Werte für die erste Zeile zurückgegeben werden können, sollten die empfangenden Host-Variablen grundsätzlich vor der Ausführung des *SELECT … INTO-Statements* initialisiert werden.

Der *SELECT … INTO-Befehl* kann nur in statischem SQL eingesetzt werden.

Im folgenden Beispiel wird die nächste Auftragspositionsnummer. ermittelt. Die höchste Positionsnummer (Max(POSNR)) für eine vorgegebene Firma und Bestellnummer wird um 1 erhöht. Sofern für den angegebenen Auftrag noch keine Position vorhanden sein sollte, wird ein NULL-Wert ausgegeben.

Durch die skalare Funktion COALESCE wird ein NULL-Wert durch einen Unterlassungswert ersetzt – d. h., sofern keine Auftragsposition gefunden wird, wird die Position 1 ausgegeben.

```
D ParFirma        S               3P 0 inz(10)
D ParBestNr       S              15A
inz('BNR-20120224/01')

D NxtPos          S               3U 0
*------------------------------------------------------------
------------
 /Free
   //Ermitteln nächste Auftragsposition
     Clear NxtPos;                    //Initialisieren von
Host-Variablen

     EXEC SQL  Select  Coalesce(Max(BestPos) + 1, 1)
                  into :NxtPos
                  from AuftrPosX
                  Where     Firma   = :ParFirma
                        and BestNr  = :ParBestNr;

     Dsply NxtPos;
 /End-Free
 ************ Quelle EMBSELIN1
```

Beispiel SELECT ... INTO-Statement

3.2.5.7.3.3 SQL-Befehl VALUES ... INTO

Das *VALUES … INTO-Statement* kann in statischem SQL als Syntaxalternative für *SELECT … INTO* verwendet werden.

Das komplette SELECT-Statement wird in Klammern nach dem Befehl VALUES hinterlegt. Die Ausgabevariablen müssen in der INTO-Anweisung nach dem Select-Statement angegeben werden.

Die Anzahl der Host-Variablen (bzw. die Anzahl der Unterfelder in Host-Strukturen) muss der Anzahl der ausgewählten Spalten entsprechen. Die Host-Variablen müssen zumindest mit einem kompatiblen Datentyp definiert worden sein.

Die Angabe von Indikator-Variablen ist optional bzw. nur dann erforderlich, wenn NULL-Werte ausgegeben werden könnten.

3.2.5
Seite 66

```
/Free
  //Syntax
  Exec SQL Values (Select Col1, Col2, ... ColN
                   From ... )
       Into :HostVar1 :IndHostVar1, :HostVar2 ... :HostVarN;
```

Syntax VALUES ... INTO Statement

Im Gegensatz zum *SELECT…INTO-Befehl* kann das *VALUES … INTO-Statement* auch dynamisch aufbereitet werden.

Im folgenden Beispiel wird geprüft, ob in der Auftragspositionsdatei (AUFTRPOS) für die vorgegebene Firma und die vorgegebene Bestellnummer mindestens eine Position mit Status kleiner 70 vorhanden ist. Durch die Angabe von FETCH FIRST ROW ONLY wird sichergestellt, dass maximal eine Zeile zurückgegeben wird. Sofern mindestens eine Position mit einem Status unter 70 vorhanden ist, wird 1 in die Host-Variable AnzPos ausgegeben.

```
D ParFirma        S               3P 0 inz(10)
D ParBestNr       S              15A   inz('BNR-20120224/01')

D MaxStatus       S               2A   inz('70')

D AnzPos          S              10I 0
D IndAnzPos       S               5I 0
 *---------------------------------------------------------
 /Free
   //Prüfen mindestens eine Auftragsposition mit Status
   < 70 vorhanden
     Clear AnzPos;
     EXEC SQL  Values (Select   1
                         from  AuftrPosX p
                         where    Firma  = :ParFirma
                              and BestNr = :ParBestNr
                              and Status < :MaxStatus
                         Fetch first Row Only)
                  Into :AnzPos :IndAnzPos;
     ...
     Dsply AnzPos;
     *INLR = *ON;
     Return;
 /End-Free
 ********** Quelle EMBVALIN1
```

VALUES ... INTO-Beispiel in statischem SQL

3.2.5.8 Dynamisches SQL ohne Cursor

Mit Hilfe von Host-Variablen können viele SQL-Statements sehr variabel gestaltet werden. Host-Variablen können in Verbindung mit Spaltenwerten in Tabellen/Dateien/Views oder als Ausgabewerte verwendet werden.

Ist jedoch die Tabelle/Datei/View und/oder die Bibliothek zur Kompilierungszeit noch nicht bekannt, muss dynamisches SQL verwendet werden.

Beispiel:

Zu Beginn jedes Monats sollen alle Bewegungsdaten, die älter als drei Monate sind, in eine Tabelle/Datei gespeichert werden. Der Name der Sicherungsdatei wird ermittelt und setzt sich aus dem Kürzel BW, dem Jahr und Monat der zu sichernden Daten zusammen – d. h., der Dateiname für die Sicherung im Januar 2012 für November 2011 wird BW201111. Der Name der Sicherungsdatei ändert sich somit jeden Monat.

Dynamisches SQL ist allerdings nicht auf SQL-Statements, in denen Dateien/Tabellen/Views variabel verwendet werden müssen, beschränkt. Vielmehr kann dynamisches SQL für jedes beliebige SQL-Statement verwendet werden.

Anmerkung:

Aus Performancegründen, insbesondere weil beim dynamischen SQL zur Laufzeit zusätzliche Schritte ausgeführt werden müssen (z. B. Syntax-Prüfung, Konvertierung in ausführbares SQL-Statement, komplette Optimierung FULL OPEN), sollte statisches SQL dem dynamischen vorgezogen werden.

Für dynamisches SQL sind die folgenden Schritte erforderlich:

Aufbau des auszuführenden SQL-Statements (Statement-String)
Das SQL-Statement muss zur Laufzeit zunächst als Text-String aufbereitet werden. Dieser aufbereitete Text wird als **Statement-String** bezeichnet.

Wenn in dem Statement-String variable Werte verwendet werden, dann müssen diese entweder innerhalb des Texts aufgelöst werden oder – sofern es sich bei den variablen Werte nicht um Tabellen/Dateien/Views oder Bibliotheken handelt – sie können als Parameter-Marker angegeben werden.

Werden die Variablen aufgelöst, so müssen alphanumerische Werte und Datumswerte in Hochkommata eingeschlossen werden. Einfache Hochkommata müssen dabei doppelt angegeben werden. Numerische Werte werden nicht in Hochkommata eingeschlossen, müssen jedoch, um eingebunden werden zu können, mit Hilfe der Built-In-Function **%CHAR**() konvertiert werden.

Datumswerte sollten in einem Datumsformat mit einem vierstelligen Jahr (*ISO, *EUR, *USA oder *JIS) aufbereitet werden. Sofern Unicode- (Datentyp C) oder andere DBCS-Werte (Datentyp G) eingebunden werden sollen, sollte der Statement-String entweder in Unicode (Datentyp C) oder grafisch (Datentyp G) aufbereitet werden.

Wenn Parameter-Marker verwendet werden, sollte anstelle der variablen Werte ein Fragezeichen (**?**) angegeben werden. Im Gegensatz zum direkten Einbinden der variablen Werte innerhalb des Strings, muss bei Verwendung von Parameter-Markern keine Rücksicht auf den Datentyp der Variablen/Werte genommen werden, d. h., weder die Einbettung in Hochkommata noch eine Konvertierung in Unicode ist erforderlich.

Konvertierung in ausführbares SQL-Statement (Prepared Statement)
Der aufbereitete Statement-String muss in ein ausführbares SQL-Statement konvertiert werden. Dabei spielt es zunächst keine Rolle, ob in dem String Parameter-Marker enthalten sind oder nicht. Das konvertierte SQL Statement wird als **Prepared Statement** bezeichnet.

Bei der Konvertierung in ein ausführbares SQL-Statement wird zunächst geprüft, ob es sich um ein gültiges SQL-Statement handelt (Syntax-Prüfung). Sofern das SQL-Statement fehlerhaft ist, wird ein entsprechender SQLCODE und SQLSTATE in der SQLCA ausgegeben. Im Programm sollte nach jedem SQL-Statement entweder der SQLCODE oder der SQLSTATE geprüft werden.

Nach der Syntax-Prüfung wird der Zugriffsplan (Access Plan) für das konvertierte SQL-Statement festgelegt.

Ausführung des konvertierten SQL-Statements
Nach der erfolgreichen Konvertierung wird das Prepared Statement ausgeführt.

Sofern mit Parameter-Markern gearbeitet wurde, werden die variablen Werte an dieser Stelle ersetzt.

Wenn das SQL-Statement zum ersten Mal nach der Konvertierung ausgeführt wird, werden die im Zugriffsplan beschriebenen temporären Objekte erstellt und mit Daten gefüllt – d. h., der ODP (Open Data Path) wird geöffnet.

Wird das SQL-Statement nach der Konvertierung mehrfach (mit unterschiedlichen) variablen Werte ausgeführt, kann der geöffnete Datenpfad wiederverwendet werden – d. h., lediglich die Daten werden aktualisiert.

Zur Ausführung von dynamischem SQL stehen die folgenden SQL-Befehle zur Verfügung

- PREPARE
  Konvertiert den Text in ein ausführbares SQL-Statement – d. h. das Prepared Statement wird erstellt.
- EXECUTE
  Führt das konvertierte SQL-Statement (Prepared Statement) aus. Das einmal konvertierte SQL-Statement kann mehrfach mit unterschiedlichen variablen Werten ausgeführt werden.
- EXECUTE IMMEDIATE
  Führt die Befehle PREPARE und EXECUTE in Kombination aus. Dieses SQL-Statement kann verwendet werden, sofern das SQL-Statement komplett, d. h. ohne Parameter-Marker aufbereitet wird.

3.2.5.8.1 SQL-Befehl EXECUTE IMMEDIATE

Der EXECUTE IMMEDIATE-Befehl kann ausgeführt werden, sofern das SQL-Statement komplett ist, ohne Parameter-Marker aufbereitet und nur einmalig ausgeführt wird. Der Befehl EXECUTE IMMEDIATE kann nicht in Verbindung mit einem SELECT-Statement ausgeführt werden.

Der aufbereitete String wird an das SQL-Statement als Host-Variable übergeben. Dabei spielt es keine Rolle, ob die Host-Variable mit dem Datentyp A (= alphanumerisch / Single Byte Character Set), C (= Unicode / Double Byte Character Set) oder G (= grafisch / Double Byte Character Set) definiert wurde.

```
/Free
  MyHostVar = 'SQL-Statement ohne Parameter-Marker' ;

  //Syntax
  Exec SQL  Execute Immediate :MyHostVar;
```

Syntax SQL-Statement EXECUTE IMMEDIATE

Beim folgenden Beispiel handelt es sich um ein kleines Reorganisationsprogramm. An das Programm werden das Jahr (für das die Daten gesichert werden sollen), die Tabelle/Datei (aus der die Daten gesichert werden sollen) und der Name des Felds mit dem Vergleichsdatum übergeben. Die Bibliothek, in der sich die Tabelle/Datei befindet, kann als optionaler Parameter übergeben werden. Sofern die Bibliothek nicht übergeben werden kann, wird die Tabelle/Datei in der Bibliotheksliste gesucht.

Ergänzung 21/2012

3.2.5
Seite 70

Basierend auf den übergebenen Parameterwerten werden die qualifizierten Namen der Originaldatei sowie der Sicherungsdatei (= Bibliothek SAVELIB + / + Original-Name + Jahr) ermittelt.

Zunächst soll die Tabelle, sofern sie bereits existiert, gelöscht werden. Hierfür wird zunächst das DROP-Statement als Text aufbereitet und anschließend mit dem Befehl EXECUTE IMMEDIATE ausgeführt. Nach Ausführung wird der SQLCODE dahingehend geprüft, ob EXECUTE IMMEDIATE erfolgreich ausgeführt wurde. Sofern die Tabelle nicht vorhanden war, wird der SQLCODE -204 ausgegeben, der aus der Fehlerbehandlung ausgeschlossen wird.

Anschließend wird die Sicherungstabelle erstellt und mit Daten gefüllt. Das SQL-Statement CREATE TABLE wird als Text aufbereitet und mit dem SQL-Befehl EXECUTE IMMEDIATE ausgeführt.

Sofern die Sicherungstabelle erfolgreich erstellt wurde, werden die gesicherten Daten aus der Original-Datei/Tabelle gelöscht – d. h., das DELETE TABLE-Statement wird als Text aufbereitet und anschließend mit EXECUTE IMMEDIATE ausgeführt.

```
D SaveHistory     PI
D   ParYear                      4P 0
D   ParFile                    128A   Varying Const
D   ParField                   128A   Varying Const
D   ParLib                     128A   Varying Const Options(*NoPass)

D SaveLibFile     S             257A   Varying
D BaseLibFile     S             257A   Varying

D StringCmd       S            1024A   Varying
 ******************************************************************************
/Free
    *INLR = *ON;

   //Aufbereiten Name der Sicherungsdatei
   If     %Parms >= %ParmNum(ParLib)
      and %Len(%Trim(ParLib)) > *Zeros
      and %Trim(ParLib) <> '*LIBL';
      BaseLibFile = %Trim(ParLib) + '/';
   EndIf;

   BaseLibFile += %Trim(ParFile);
   SaveLibFile  = 'SAVELIB/' + %Trim(ParFile) + %EditC(ParYear: 'X');
```

```
   //Löschen Sicherungsdatei, wenn vorhanden
   StringCmd = 'Drop Table ' + %Trim(SaveLibFile);
   EXEC SQL  Execute Immediate :StringCmd;
   If     SQLCODE  < *Zeros   //Fehler
      and SQLCODE <> -204;    //Objekt nicht vorhanden
      //Fehler-Behandlung
      Return;
   EndIf;

   //Erstellen Sicherungsdatei mit Daten
   StringCmd = 'Create Table '+ %Trim(SaveLibFile) +
                  ' as (Select * From  ' + %Trim(BaseLibFile) +
                         ' where Year(' + %Trim(ParField) + ') = ' +
                                          %Char(ParYear)  +
               ' ) With Data ';
   EXEC SQL  Execute Immediate :StringCmd;
   If SQLCODE < *Zeros;
      //Fehlerbehandlung
      Return;
   EndIf;

   //Löschen Daten aus Originaldatei
   StringCmd = 'Delete From ' + %Trim(BaseLibFile) +
               ' where Year(' + %Trim(ParField) + ') = ' + %Char(ParYear);
   EXEC SQL  Execute Immediate :StringCmd;
   If SQLCODE < *Zeros;
      //Fehlerbehandlung
      Return;
   EndIf;

   Return;
/End-Free
********** Quelle EMBDYNIM1
```

Beispiel Execute Immediate – Reorganisation

Ergänzung 21/2012

3.2.5.8.2 SQL-Befehle PREPARE und EXECUTE

Sofern das gleiche dynamisch aufbereitete SQL-Statement mehrfach unverändert oder mit unterschiedlichen variablen Werten ausgeführt werden soll, oder aber sofern Daten von dem dynamischen Statement ausgegeben werden sollen, kann der Befehl EXECUTE IMMEDIATE nicht verwendet werden.

Stattdessen müssen die SQL-Befehle **PREPARE und EXECUTE** ausgeführt werden.

3.2.5.8.1.1 SQL-Befehl PREPARE

Bei dem SQL-Befehl PREPARE wird der aufbereitete String (Statement-String) mit oder ohne Parameter-Marker in ein ausführbares SQL-Statement konvertiert.

Konnte der Text nicht in ein ausführbares SQL-Statement konvertiert werden (z. B. wegen eines Syntax-Fehlers), erfolgt kein (Programm-)Abbruch. Es wird lediglich ein Fehlercode in der SQLCA ausgegeben. Nach Ausführung des SQL-Statements PREPARE sollte deshalb dieser Fehlercode – also der Inhalt in SQLCODE oder SQLSTATE – geprüft werden und für den Fehlerfall entsprechende Aktionen im Programm vorgesehen werden.

Für das ausführbare SQL-Statement muss ein **innerhalb der Quelle eindeutiger Name** festgelegt werden. Der Name muss auch dann eindeutig sein, wenn in der Quelle mehrere interne oder exportierte Prozeduren hinterlegt sind, in denen dynamische SQL-Statements verwendet werden. Da es sich bei diesem eindeutigen Namen um keine Host-Variable handelt, darf auch kein führender Doppelpunkt angegeben werden.

```
/Free
   MyHostVar = 'SQL-Statement mit oder ohne Parameter-
   Marker';

   //Syntax
   Exec SQL Prepare MyDynSQL from :MyHostVar;

   If SQLCODE < *Zeros;        //Fehler
      //Fehlerbehandlung
   EndIf;
```

Syntax SQL-Befehl PREPARE

3.2.5.8.1.2 SQL-Befehl EXECUTE

Mit dem SQL-Statement EXECUTE wird ein mit Hilfe des SQL-Statements PREPARE konvertierter SQL-Befehl (Prepared Statement) ausgeführt. Dabei wird der im Befehl PREPARE festgelegte Name für das konvertierte SQL-Statement angegeben.

Sofern Parameter-Marker zum Setzen oder Empfangen von Werten verwendet wurden, müssen die Host-Variablen, die die variablen Werte enthalten bzw. empfangen, im EXECUTE Statement in einer USING-Anweisung aufgelistet werden.

Die Anzahl der Parameter-Marker muss mit der Anzahl der angegebenen Host-Variablen übereinstimmen. Der Wert der ersten Variable wird dem ersten Parameter-Marker zugeordnet, der Wert der zweiten Variable dem zweiten Parameter-Marker usw. Anstatt einzelne Host-Variablen aufzulisten, können auch einfache Host-Strukturen verwendet werden. Sofern Host-Variablen zum Empfangen von Ausgabewerten verwendet werden, können zusätzlich Indikator-Variablen zur Prüfung von NULL-Werten angegeben werden.

Anmerkung:

Bei Host-Variablen, die in der Using-Anweisung angegeben werden, darf es sich nicht um Parameter-Felder handeln, die mit Constant Reference (Schlüsselwort CONST) übergeben wurden.

Dabei spielt es keine Rolle, ob die Host-Variablen im dynamischen SQL nur zum Setzen von variablen Werten oder auch zum Empfangen von Ausgabewerten verwendet werden.

Um festzustellen, ob ein aufbereiteter SQL-Befehl erfolgreich ausgeführt wurde, sollte der im SQLCODE oder SQLSTATE ausgegebene Fehlercode geprüft werden.

```
/Free
   //Syntax ohne Parameter-Marker
   Exec SQL  Execute MyDynSQL;

   If SQLCODE < *Zeros;        //Fehler
      //Fehlerbehandlung
   EndIf;

   //Syntax mit Parameter-Marker/Host-Variablen
   Exec SQL  Execute  MyDynSQL
               Using :HostVar1, :HostVar2 :IndVar2,
               ... :HostVarN;

   If SQLCODE < *Zeros;        //Fehler
      //Fehlerbehandlung
   EndIf;
```

Syntax SQL-Befehl EXECUTE

3.2.5.8.1.3 Dynamisches SQL mit PREPARE und EXECUTE – ein Beispiel

Das folgende Beispiel zeigt eine Prozedur zum Sichern der Tagesumsätze für eine übergebene Kunden- und Artikelnummer. Die Ausgabe erfolgt in eine Arbeitsdatei, die sich in der Bibliothek QTEMP befindet. Die Arbeitsdatei, deren Namen sich aus WF und dem Tagesdatum zusammensetzt, wird erstellt, sofern sie noch nicht angelegt worden ist.

Das INSERT-Statement wird nur einmalig, nämlich beim ersten Aufruf aufbereitet, jedoch beim ersten und bei jedem weiteren Aufruf ausgeführt. Sofern ein SQL-Statement fehlerhaft beendet wurde, wird *OFF als Rückgabewert ausgegeben. Sofern die Prozedur fehlerfrei ausgeführt worden ist, wird *ON ausgegeben.

```
P SavTagesUmsatz  B

D SavTagesUmsatz  PI              N
D   ParKunde                    15A    Const
D   ParArtikel                  22A    Const

D HostKunde       S             15A
D HostArtikel     S             22A
D WrkTable        S             10A
```

```
D WrkTablePrv     S             10A              Static
D CmdSQLCreate    S           1024A    Varying
D CmdSQLInsert    S           1024A    Varying
D IsPrepInsert    S              N              Static
 *-------------------------------------------------------
 /Free
   HostKunde   = %Trim(ParKunde);
   HostArtikel = %Trim(ParArtikel);

   //Leere Arbeitsdatei in QTEMP erstellen (sofern nicht
     vorhanden)
   WrkTable = 'WF' + %Char(%Date(): *ISO0);

   If WrkTable <> WrkTablePrv;
      CmdSQLCreate = 'Create Table QTEMP/' + WrkTable +
                       ' as (Select * From Umsatz) +
                     With No Data';
      Exec SQL Execute Immediate :CmdSQLCreate;
      If     SQLCODE < *Zeros     //Fehler
         and SQLCODE <> -601;     //Tabelle bereits vorhanden
         Return *OFF;
      EndIf;
      WrkTablePrv = WrkTable;
   EndIf;

   //Aufbereiten INSERT-Statement (sofern noch nicht erfolgt)
   If Not IsPrepInsert;
      CmdSQLInsert = 'Insert into QTEMP/' + WrkTable    +
                     ' Select * from Umsatz       +
                         Where     KundeNr   = ? +
                               and ArtikelNr = ? +
                              and VerkDatum = Current_Date';
      Exec SQL Prepare DynSQLInsert From :CmdSQLInsert;
      If SQLCODE < *Zeros;
         Clear IsPrepInsert;
         Return *OFF;
      EndIf;
      IsPrepInsert = *On;
   EndIf;
```

```
    Exec SQL Execute DynSQLInsert Using :HostKunde,
    :HostArtikel;
    If SQLCODE < *Zeros;
       Return *OFF;
    EndIf;

    Return *ON;
  /End-Free
P SavTagesUmsatz  E
 ********** Quelle EMBDYNPE1
```

Beispiel SQL Commands PREPARE und EXECUTE

3.2.5.9 SQL-Befehl VALUES ... INTO in dynamischem SQL

Das *VALUES … INTO-Statement* kann dazu verwendet werden, um einen einzelnen Datensatz zu ermitteln. Im Gegensatz zum *SELECT … INTO-Statement* kann das *VALUES … INTO-Statement* dynamisch aufbereitet werden.

Da aus dem *VALUES … INTO-Statement* Informationen in Host-Variablen ausgegeben werden müssen, müssen die SQL-Befehle PREPARE und EXECUTE in Verbindung mit einer USING-Anweisung verwendet werden.

Das folgende Beispiel zeigt eine Funktion, in der für jede beliebige Datei/Tabelle die Anzahl der Datensätze ermittelt wird. Der Datei-/Tabellen-Name wird als Parameter übergeben. Die Bibliothek / das Schema kann als optionaler Parameter ebenfalls übergeben werden.

Das *SQL-SELECT-Statement* wird innerhalb des *VALUES … INTO-Statements* hinterlegt. Für die Ausgabe wird ein Parameter-Marker nach INTO angegeben.

Das fertige *VALUES … INTO-Statement* wird zunächst über den SQL-Befehl PREPARE in einen ausführbaren SQL-Befehl konvertiert und anschließend mit dem SQL-Befehl EXECUTE ausgeführt. Die Host-Variable RTNNBRROWS, in die das Ergebnis ausgegeben werden soll, wird im EXECUTE-Befehl in der USING-Anweisung angegeben.

Im Fehlerfall wird der negative SQL-CODE an die rufende Prozedur zurückgegeben. Wenn die Anzahl der Zeilen ordnungsgemäß ermittelt werden konnte, wird dieser Wert an die rufende Prozedur ausgegeben.

```
 ******************************************************************************
 * Procedure Name:   NbrOfRows
 * Purpose:          Ermittelt die Anzahl Zeilen/Datensätze in Tabelle
 * Returns:          Anzahl der Zeilen
 * Parameter:        ParTable  => Tabelle/phys.Datei (SQL oder System-Name)
 * Parameter:        ParSchema => Schema/Bibliothek
 ******************************************************************************
P NbrOfRows       B

D NbrOfRows       PI            10I 0
D  ParTable                    128A   Varying Const
D  ParSchema                   128A   Varying Const Options(*NoPass)

D RtnNbrRows      S             10I 0

D CmdSQL          S           1024A   Varying
 *----------------------------------------------------------------
 /Free
   CmdSQL = 'Values( Select Count(*) from ';

   If     %Parms >= %ParmNum(ParSchema)
      and %Len(%Trim(ParSchema)) > *Zeros;
      CmdSQL += %Trim(ParSchema) + '/';
   EndIf;

   CmdSQL += %Trim(ParTable) + ' ) into ?';

   Exec SQL  Prepare DynSQL from :CmdSQL;
   If SQLCODE < *Zeros;
      Return SQLCODE;
   EndIf;

   Exec SQL  Execute DynSQL using :RtnNbrRows;
   If SQLCOD < *Zeros;
      Return SQLCODE;
   EndIf;

   Return RtnNbrRows;
 /End-Free
P NbrOfRows       E
 ***************** Quelle EMBDYNVI1
```

Values … into in dynamischem SQL

3.2.5.10 Definition und Verwendung der Cursors

Für die Verarbeitung von Ergebnissen aus den SELECT-Statements können die bislang gezeigten Techniken nicht verwendet werden, da jede Zeile gelesen und einzeln verarbeitet werden muss. So sollen z. B. die selektierten Zeilen am Bildschirm in einer klassischen Subfile angezeigt, in ein HTML-Dokument ausgegeben oder auch in eine Printer-File geschrieben werden Um SELECT-Statements verarbeiten zu können, muss ein Cursor verwendet werden. Für die Cursor-Verarbeitung sind die folgenden Schritte erforderlich:

SQL-Befehl DECLARE CURSOR
Im *DECLARE CURSOR-Statement* wird das zu verarbeitende *SELECT-Statement* vorgegeben und einem innerhalb der Quelle eindeutigen Cursor zugeordnet werden. Die Anzahl der aus dem *SELECT-Statement* ausgegebenen Zeilen kann durch die Angabe von WHERE-Bedingungen in Verbindung mit Host-Variablen eingeschränkt und variabel gestaltet werden.

Das *DECLARE CURSOR-Statement* entspricht beim native I/O der Definition einer Datei in den F-Bestimmungen, die User Controlled geöffnet werden kann. Beim Native I/O kann die Anzahl der zu lesenden Datensätze nur durch die Angabe der entsprechenden Zugriffswege (DDS-beschriebene, logische Dateien oder SQL Indices) in den F-Bestimmungen beeinflusst werden. Liegt kein entsprechender Zugriffsweg vor, müssen die nicht benötigten Datensätze mit Hilfe des OpCodes ITER überlesen werden.

Cursor können auf zwei Arten definiert werden:

- Serieller Cursor
  Bei einem seriellen Cursor (Unterlassungswert) können die aus dem SELECT-Statement ausgegebenen Zeilen (Result Set) nur vorwärts gelesen werden.
  Ein serieller Cursor kann mit einer Datei, die in den F-Bestimmungen als IP (Input Primary) definiert wurde, verglichen werden.
- Scroll Cursor
  Bei einem Scroll Cursor können die aus dem SELECT-Statement ausgegebenen Zeilen, beliebig vorwärts und rückwärts gelesen werden. Ebenso ist es möglich, den Anfang oder das Ende der Daten zu positionieren oder die gleiche Zeile erneut einzulesen. Außerdem kann innerhalb des Result-Sets eine vorgegebene Anzahl von Zeilen vorwärts oder zurück positioniert werden.
  Es ist jedoch nicht möglich, innerhalb des Result Sets ein oder mehrere Schlüsselfelder zu positionieren.
  Ein Scroll Cursor kann mit einer beliebigen, in den F-Bestimmungen definierten Input-Datei verglichen werden, die User Controlled geöffnet wird.

SQL-Befehl OPEN
Mit dem Open-Befehl wird der Cursor geöffnet – d. h., das hinterlegte SELECT-Statement wird mit den in den Host-Variablen hinterlegten Werten ausgeführt und auf die erste Ergebniszeile positioniert.

Die Ausführung des OPEN-Befehls ist die zeitaufwändigste Aktion bei der Cursor-Verarbeitung, da an dieser Stelle die Optimierung erfolgt. Dabei wird der Zugriffsplan erstellt oder validiert. Die im Zugriffsplan beschriebenen temporären Objekte werden generiert und – basierend auf den beschriebenen Zugriffsmethoden und den ermittelten Zugriffswegen – mit Daten gefüllt.

Bei der Verarbeitung mit native I/O kann das *SQL-OPEN-Statement* mit dem (manuellen) Öffnen der Datei durch den OpCode OPEN und der Positionierung des Datei-Pointers auf den ersten Datensatz durch den OpCode SETLL verglichen werden.

SQL-Befehl FETCH … FROM CURSOR
Mit dem *FETCH-Statement* kann entweder eine einzelne Zeile in Host-Variablen oder -Strukturen oder können auch mehrere Zeilen in eine Mehrfach- oder Array-Datenstruktur ausgegeben werden.

Bei einem seriellen Cursor kann immer nur die nächste Zeile bzw. der nächste Block an Zeilen gelesen werden. Die Reihenfolge bzw. die Sortierung, in der die Zeilen ausgegeben werden, wird in der ORDER BY-Anweisung innerhalb des *SELECT-Statement* beim DECLARE CURSOR festgelegt. Wurde keine *ORDER BY-Anweisung* angegeben, so wird die Reihenfolge der ausgegebenen Daten vom verwendeten Zugriffsweg bestimmt.

Bei einem Scroll Cursor kann die nächste zu verarbeitende Zeile oder der nächste Block an Zeilen im *FETCH-Statement* (z. B. FETCH NEXT oder FETCH PRIOR) angegeben werden – d. h., innerhalb des Result-Sets kann vorwärts und rückwärts gelesen werden.

Das *FETCH-Statement* kann damit mit den RPG-OpCodes READ, READE, READP und READPE verglichen werden. Im Gegensatz zu embedded SQL kann beim native I/O jedoch immer nur eine Zeile nach der anderen verarbeitet werden.

Wie auch beim native I/O muss nach Ausführung des FETCH-Statements geprüft werden, ob das Ende der Daten erreicht wurde. Während mit native I/O die Prüfung entweder über die Gleich-Bezugszahl (altes RPG!!!) oder durch die Built-In-Function *%EOF()* erfolgt, muss bei embedded SQL der SQLCODE oder der SQLSTATE aus der SQLCA geprüft werden.

Ergänzung 21/2012

3.2.5
Seite 80

SQL-Befehl CLOSE

Nachdem alle Zeilen des Result-Sets verarbeitet wurden, sollte der Cursor über den SQL-Befehl CLOSE explizit geschlossen werden.

Sofern erneut Daten in Verbindung mit anderen Host-Variablen-Werten verarbeitet werden sollen, muss zunächst der geöffnete Cursor geschlossen werden. Ein geöffneter Cursor kann nicht erneut geöffnet werden – d. h., es werden weder die Daten, die auf dem Result-Set basieren, auf neue Werte aktualisiert noch wird auf die erste Zeile des Result-Sets positioniert.

Bei der Verarbeitung mit native I/O kann das SQL-Statement CLOSE mit dem RPG-OpCode CLOSE verglichen werden.

In der folgenden Tabelle werden die Schritte, die mit native I/O zur Verarbeitung von mehreren Datensätzen erforderlich sind, den entsprechenden embedded SQL-Befehlen gegenübergestellt.

Native I/O		Embedded SQL		
Befehl	Beschreibung	Befehl	Beschreibung	
F-Bestimmungen	Definition der physischen und logischen Dateien bzw. SQL Tables/Views und Indices.	Declare Cursor	Definition des auszuführenden SQL-Statements	
UserOpen	Dateien werden direkt geöffnet, es sei denn das Schlüssel-Wort UserOpen wurde angegeben.		Serial	Unterlassungs-Wert, kann mit dem RPG-Zyklus verglichen werden.
			Scroll	Innerhalb des Ergebnisses kann vorwärts und rückwärts gelesen werden.
Open	Dateien werden Benutzer-kontrolliert geöffnet.	Open Cursor	Optimierung, Erstellung der temporären Objekte, Ermitteln der Daten (Erstellung des ODP) und Positionierung am Anfang der Daten	
SetLL/SetGT	Positionierung am Anfang oder Ende der zu lesenden Sätze.		Serial	Cursor muss vor dem Öffnen geschlossen sein.
Read/ReadE	Einlesen des nächsten Satzes.	Fetch Rows	Serial	Einlesen der nächsten Zeile(n).
ReadP/ReadPE	Einlesen des vorhergehenden Satzes.		Scroll	Einlesen der nächsten, vorhergehenden, ersten und letzten Zeile.
Close	Dateien werden Benutzer-kontrolliert geschlossen	Close Cursor	Serial	Cursor muss vor erneutem Öffnen geschlossen sein.

Gegenüberstellung Record Level Access und Embedded SQL

3.2.5.10.1 SQL-Befehl DECLARE CURSOR

Über das DECLARE CURSOR-Statement wird das zu verarbeitende **SELECT-Statement** definiert und einem **eindeutigen Cursor** zugeordnet.

Innerhalb einer Quelle, aber auch innerhalb einer Prozedur können mehrere unabhängige Cursor definiert werden. Für jeden Cursor muss ein eindeutiger Name – innerhalb der Quelle – vergeben werden. Doppelte Cursor-Namen sind auch in Quellen nicht erlaubt, in denen nur interne und exportierte Prozeduren (Schlüsselwort NOMAIN in den H-Bestimmungen) hinterlegt sind.

Anmerkung:

Das *DECLARE CURSOR-Statement* muss **innerhalb der Quelle vor allen anderen** den Cursor betreffenden SQL-Statements (OPEN, FETCH, CLOSE) angegeben werden.

Der SQL-Precompiler durchsucht die Quelle nach SQL-Befehlen – ohne Rücksicht auf die Reihenfolge, in der die SQL-Befehle ausgeführt werden. In RPG werden die Cursor in den C-Bestimmungen definiert/deklariert. In anderen Programmiersprachen, so auch in der SQL-Programmierung müssen die *DECLARE CURSOR-Statements* in der sogenannten DECLARE-Sektion hinterlegt werden, die mit den D-Bestimmungen verglichen werden kann. Diese Anordnung – d. h., DECLARE vor OPEN – wird auch bei embedded SQL vorausgesetzt.

Sofern diese Regel verletzt wird, wird die Kompilierung vom SQL-Precompiler abgebrochen.

Wird z. B. ein *DECLARE CURSOR-Statement* in der Initialisierungs-Subroutine *INZSR, die am Ende der Quelle platziert ist, hinterlegt und das *OPEN-Statement* im Hauptprogramm ausgeführt, kann das Programm nicht kompiliert werden.

3.2.5
Seite 82

3.2.5.10.1.1 SQL-Befehl DECLARE CURSOR für serielle Cursor

Sofern im *DECLARE Cursor Statement* keine weiteren Angaben gemacht werden, wird ein **serieller Cursor** definiert – d. h., bei der Verarbeitung können die ausgegebenen Zeilen nur vorwärts gelesen und verarbeitet werden.

```
/Free
   //Syntax
   EXEC SQL  DECLARE MyCursor CURSROR
                 FOR Select ... ;
```

Syntax DECLARE CURSOR – serieller Cursor

Im folgenden Beispiel wird ein serieller Cursor zur Ermittlung der kumulierten Umsätze innerhalb eines vorgegebenen Zeitraums, für einen einzelnen Kunden oder für alle Kunden definiert.

Der Zeitraum – also das Von- und Bis-Datum – sowie der Kunde werden über Host-Variablen in das *SELECT-Statement* eingebunden. Durch die *CASE-Anweisung* wird gesteuert, ob die Umsätze für einen einzelnen Kunden oder für alle Kunden ermittelt werden sollen.

Die *CUBE-Klausel* in der *GROUP BY-Anweisung* bewirkt, dass zusätzlich zu den Umsätzen auf Kundenebene auch der Gesamtumsatz ausgegeben wird.

Die Angabe OPTIMIZE FOR 10 ROWS gibt das Optimierungsziel *FIRSTIO vor.

```
/Free
  Exec SQL Declare CsrC1 Cursor
           For Select KundeNr, Sum(Wert)
                  from Umsatz
                  where     VerkDatum between :HostVonDatum
                                          and :HostBisDatum
                        and KundeNr = Case When
                        :HostKunde > ' '
                                           Then :HostKunde
                                           Else KundeNr End
                  Group By Cube(KundeNr)
                  Order BY KundeNr
               Optimize For 10 Rows;
 ...
 /End-Free
 ********** Quelle EMBCSRSS1
```

Beispiel DECLARE CURSOR – serieller CURSOR

3.2.5.10.1.2 SQL-Befehl DECLARE CURSOR für Scroll („rollende“) Cursor

Beim Scroll Cursor, bei dem innerhalb der ausgegebenen Zeilen (Result-Set) vorwärts und rückwärts gelesen werden kann, muss lediglich zwischen dem individuellen Cursor-Namen und dem Schlüssel-Wort CURSOR SCROLL eingefügt werden.

Sofern die Datensätze über den Scroll Cursor geändert werden sollen, muss vor SCROLL zusätzlich DYNAMIC eingefügt werden.

```
/Free
  //Syntax - Read Only Scroll Cursor
  EXEC SQL   DECLARE MyCursor SCROLL CURSOR
                 FOR Select ...;

  //Syntax - Updatable Scroll Cursor
  EXEC SQL   DECLARE MyCursor DYNAMIC SCROLL CURSOR
                 FOR Select ...;
```

Syntax DECLARE CURSOR – Scroll Cursor

Im folgenden Beispiel wird ein updatefähiger Scroll Cursor, durch die Angabe von DYNAMIC SCROLL CURSOR definiert.

Über diesen Cursor werden Artikelnummer, Artikelbezeichnung sowie der Preis für alle Artikel des Artikelstamms ohne Einschränkung eingelesen.

Durch die Angabe von FOR UPDATE OF wird festgelegt, dass über diesen Cursor lediglich die Spalte PREIS im Artikelstamm geändert werden kann.

```
/Free
  Exec SQL Declare CsrC02 Dynamic Scroll Cursor
              For Select  ArtikelNr, ArtikelBez, Preis
                     from ArtikelSt
                  For Update Of Preis;
```

Beispiel DECLARE CURSOR – updatefähiger Scroll Cursor

3.2.5.10.1.3 SQL-Befehl DECLARE CUSROR und Commitment-Steuerung

Per Default werden geöffnete Cursor, die unter Commitment-Steuerung verarbeitet werden, geschlossen, sobald einer der Befehle COMMIT oder ROLLBACK ausgeführt wird.

Sofern ein Cursor weder durch COMMIT noch durch ROLLBACK geschlossen werden soll, kann dies durch die Angabe von WITH HOLD in dem *DECLARE CURSOR-Statement* vor dem *SELECT-Statement* verhindert werden. Dabei spielt es keine Rolle, ob ein serieller, ein Scroll, ein updatefähiger oder ein Read Only Cursor definiert wird. WITH HOLD kann für jede Art von Cursor angegeben werden.

```
/Free
  //Syntax
  EXEC SQL   DECLARE  MyCursor ((DYNAMIC) SCROLL) CURSOR
                      WITH HOLD
                 FOR  Select ...;
```

Syntax DECLARE CURSOR and Commitment Control

Beispiel:

Über den ersten Cursor werden alle Auftragsköpfe mit einem bestimmten Liefertermin verarbeitet.

Für jeden gelesenen Auftrag müssen entweder alle Auftragspositionen oder keine Auftragsposition ausgeliefert werden. Die Auftragspositionen werden über einen zweiten Cursor verarbeitet.

Wenn alle Auftragspositionen für einen Auftrag ordnungsgemäß verarbeitet werden konnten, wird die Transaktion beendet – d. h., die Änderungen werden über COMMIT festgeschrieben. Im Fehlerfall werden die Auftragspostionen über ROLLBACK zurückgesetzt.

Sofern im DECLARE CURSOR-Statement für die Auftragsköpfe nicht WITH HOLD angegeben wurde, wird dieser Cursor beim COMMIT oder ROLLBACK ebenfalls geschlossen.

Das folgende Beispiel zeigt das DECLARE CURSOR-Statement zur Verarbeitung der Auftragsköpfe.

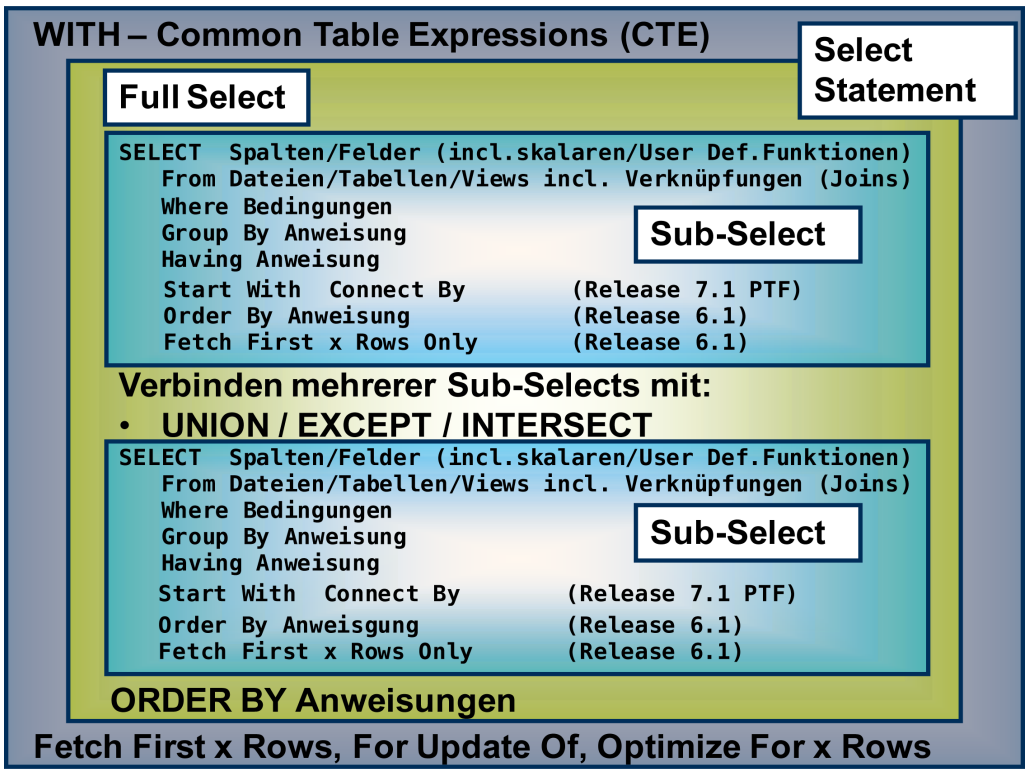
```
/Free
   Exec SQL  Declare CsrAK Cursor WITH HOLD
                 For Select Firma, BestNr
                        From AuftrKopf
                        Where LiefTerm = :HostDatum
                        Order By Firma, BestNr
                    For Read Only Optimize For 5 Rows;
```

SQL-Befehl DECLARE CURSOR ... WITH HOLD – Beispiel

3.2.5.10.1.4 SQL-Befehl SELECT in Verbindung mit Cursor-Deklarationen

Das SELECT-Statement, das innerhalb eines DECLARE CURSOR-Befehls angegeben werden kann, kann alles enthalten, was in einem SELECT-Statement erlaubt ist – inklusive der Verwendung von skalaren Funktionen, Case-Anweisungen, GROUP BY und HAVING-Anweisungen, alle Arten von Verknüpfungen, UNION/EXEPT und INTERSECT-Anweisungen, verschachtelten Sub-Selects, (rekursive) Common Table Expressions.

Die folgende Abbildung gibt einen Überblick über die Struktur des SQL-SELECT-Statements.



SQL Select-Statement

3.2.5
Seite 86

Durch die Angabe von WHERE-Bedingungen in Verbindung mit Host-Variablen, kann die Anzahl der Ergebniszeilen eingeschränkt werden.

Die Angabe von SELECT* sollte soweit wie möglich vermieden werden. Stattdessen sollten nur die tatsächlich benötigten Spalten ausgewählt werden. Wenn Spalten gezielt ausgewählt werden, ist eine erneute Kompilierung des embedded SQL-Programms bei einer Dateierweiterung nicht erforderlich. Im Gegensatz zu native I/O, bei dem immer nur der komplette Datensatz eingelesen wird, werden bei embedded SQL nur die angeforderten Informationen ermittelt und zurückgegeben. Je weniger Informationen in einer Zeile enthalten sind, desto mehr Zeilen können in einem Block ausgegeben werden – d. h., desto weniger Zugriffe auf die Datenbank sind erforderlich.

Obwohl in SELECT-Statements DDS-beschriebene logische Dateien angegeben werden können, sollte dies ebenfalls vermieden werden. Der Query-Optimizer muss ein SELECT-Statement, das DDS-beschriebene logische Dateien enthält vor der Optimierung – basierend auf den zugrundeliegenden physischen Dateien bzw. SQL-Tabellen unter Berücksichtigung der Feldauswahl, den JOIN-Anweisungen und SELECT/OMIT-Klauseln – neu schreiben. Vor Release 7.1 konnten die DDS-beschriebenen logischen Dateien nur von der klassischen Query Engine (CQE) analysiert werden.

Um die SELECT-Statements innerhalb der Programme so einfach wie möglich zu gestalten, sollten zumindest die entsprechenden SQL-Views, die für die Verknüpfung der verschiedenen Dateien/Tabellen angelegt worden sind, verwendet werden. Im Idealfall sollte sich das im embedded SQL verwendete SELECT-Statement auf ein „SELECT Feldauswahl FROM VIEW“ sowie einige individuelle WHERE-Bedingungen reduzieren lassen.

Sofern die Verarbeitung der Datensätze in einer bestimmten Reihenfolge erfolgen soll, muss dem SELECT-Statement eine ORDER BY-Anweisung hinzugefügt werden.

Am Ende des SQL-Statements können noch einige spezielle Informationen hinzugefügt werden:

FOR READ ONLY oder FOR FETCH ONLY
Durch die Angabe von FOR READ ONLY oder FOR FETCH ONLY wird der Cursor als Input-Only definiert – d. h., Update oder Delete von Zeilen über den Cursor sind nicht möglich.

Einige SQL-Statements sind von Natur aus schon READ ONLY – z. B., wenn Dateien/Tabellen miteinander verknüpft werden oder wenn eine GROUP BY-Anweisung angegeben wurde.

Bei SQL-Statements, die updatefähig sind, kann die Angabe von FOR READ ONLY die Performance verbessern, da geblockt gelesen werden kann und exklusive Lock-Situationen vermieden werden können.

FOR UPDATE oder FOR UPDATE OF Spalte1, Spalte2, … SpalteN
Sofern eine Spaltenliste angegeben wurde, können nur die angegebenen Spalten über den Cursor geändert werden. Die Spalten, die geändert werden sollen, müssen nicht zwangsläufig in der SELECT-Anweisung aufgelistet sein. Werden im Update-Statement andere als in der FOR UPDATE-Klausel angegebene Spalten verwendet, wird der Datensatz nicht aktualisiert und stattdessen ein Fehler ausgegeben.

Sofern in einem updatefähigen SELECT-Statement in der FOR UPDATE-Klausel keine Spalten-Liste angegeben wurde – oder die FOR UPDATE-Klausel fehlt, können nur die im SELECT-Statement ausgewählten Spaltenwerte geändert werden.

Wenn im SELECT-Statement eine ORDER BY-Anweisung hinterlegt wurde und über den Cursor Zeilen modifiziert oder gelöscht werden sollen, muss eine FOR UPDATE-Klausel angegeben werden.

OPTIMIZE FOR X ROWS
Über die OPTIMIZE FOR-Klausel kann das Optimierungsziel beeinflusst werden.

Dynamische SQL-Statements werden per Default mit dem Optimierungsziel *FIRSTIO optimiert, während statische SQL-Statements mit dem Optimierungsziel *ALLIO optimiert werden.

Das Optimierungsziel *FIRSTIO bedeutet, dass der erste Block an Daten so schnell wie möglich ausgegeben wird, während das Optimierungsziel *ALLIO so ausgelegt ist, dass das komplette Ergebnis so schnell wie möglich ausgegeben wird. Sofern der Optimizer nur einen suboptimalen Zugriffsweg findet, wird er sich bei *FIRSTIO eher für diesen Zugriffsweg entscheiden, während bei *ALLIO ein Table Scan (Verarbeitung aller Datensätze) bevorzugt wird.

Wird in der OPTIMIZE FOR X ROW-Klausel für X ein sehr kleiner numerischer Wert eingesetzt (z. B. 3 oder 7) wird das Optimierungsziel *FIRSTIO verwendet, bei einer sehr großen Zahl oder ALL dagegen das Optimierungsziel *ALLIO.

3.2.5.11 SQL-Befehl OPEN Cursor

Durch die Ausführung des SQL-Statements OPEN wird die Abfrage mit den zum gegenwärtigen Zeitpunkt enthaltenen Werten in den verwendeten Host-Variablen ausgeführt. Die ermittelten Datensätze/Zeilen werden in temporären Objekten bereitgestellt. Der Cursor wird auf die erste Zeile dieses Result Sets positioniert.

Der im DECLARE CURSOR-Statement festgelegte eindeutige Cursor-Name muss im OPEN-Befehl angegeben werden.

```
/Free
  //Syntax
  Exec SQL   OPEN MyCursor;
```

Syntax OPEN CURSOR

Das OPEN-Statement muss innerhalb der Quelle physisch nach dem DECLARE CURSOR-Statement angegeben werden. Beide SQL-Befehle können in unterschiedlichen Subroutinen (oder Hauptprogramm und Subroutine), hinterlegt werden, sie müssen jedoch innerhalb der gleichen Prozedur codiert werden. Wird diese Regel verletzt, kann das embedded SQL-Modul/Programm nicht erstellt werden.

Ein bereits geöffneter Cursor kann nicht erneut geöffnet werden.

Anmerkung:

Beim Versuch einen bereits geöffneten Cursor erneut zu öffnen, wird kein Fehlercode (weder SQLCODE noch SQLSTATE) ausgegeben.

Tipp:

Unmittelbar vor dem OPEN-Befehl sollte der CLOSE-Befehl ausgeführt werden. Wenn der Cursor geöffnet war, so ist er jetzt geschlossen. War der Cursor nicht geöffnet, so kann der SQLCODE (-511) oder SQLSTATE (24501) einfach ignoriert werden.

```
/Free
   Exec SQL Declare CsrC1 Cursor
            For Select ... ;

   Exec SQL Close CsrC1;
   If SQLCODE < *Zeros and SQLCODE <> -511;
      //Fehlerbehandlung
   EndIf;

   Exec SQL Open CsrC1;
   If SQLCODE < *Zeros;
      //Fehlerbehandlung
   EndIF;
```

Beispiel – Open Cursor

3.2.5.12 SQL-Befehl FETCH – Zeilen/Datensätze einlesen

Mit Hilfe des FETCH-Befehls werden die im Result-Set ausgegebenen Datensätze empfangen und in Host-Variablen oder Host-Strukturen ausgegeben.

Wenn ein serieller Cursor definiert wurde, wird mit jedem FETCH die nächste Zeile aus dem Result-Set gelesen. Bei einem Scroll Cursor wird per Default ebenfalls die nächste Zeile gelesen. Es ist jedoch durch die Angabe von diversen Schlüsselworten im FETCH-Befehl möglich, den Cursor innerhalb des Result-Sets vorwärts und rückwärts zu bewegen.

Sofern – basierend auf geänderten Host-Variablen-Werten – andere Daten verarbeitet werden sollen, muss sowohl ein serieller als auch ein Scroll Cursor zunächst geschlossen und anschließend erneut geöffnet werden.

Wurde eine Zeile ordnungsgemäß in die Host-Variablen ausgegeben, wird der SQLCODE auf 0 und der SQLSTATE auf '00000' gesetzt. Beide Variablen sind in der SQLCA hinterlegt.

Sofern das Ende des Result-Sets erreicht wurde – d. h., keine weitere Zeile mehr empfangen werden kann, wird SQLCODE 100 bzw. SQLSTATE '02000' ausgegeben.

3.2.5
Seite 90

Um eine ordnungsgemäße Verarbeitung festzustellen, sollte nicht auf SQLCODE *Zeros/0 bzw. SQLSTATE ‘00000‘ geprüft werden, da es Situationen gibt, in denen die Zeileninformationen korrekt übertragen werden, jedoch Warnungen ausgegeben werden. Bei einer Warnung für den SQLCODE sollte ein positiver Wert (<> 100) ausgegeben werden. Bei einer Warnung wird außerdem ein SQLSTATE aus der Statusklasse 01 (Position 1-2 im SQLSTATE) ausgegeben. Eine Warnung wird z. B. ausgegeben, wenn Tabellen über JOIN-Anweisungen verknüpft, und mit SELECT * alle Spalten ausgewählt werden, aber im FETCH-Statement nur eine Host-Struktur für die erste Tabelle angegeben wird.

Anmerkung:

Nach dem FETCH-Statement sollte der SQLCODE auf 100 (Last Record) und kleiner Null (Fehler) oder der SQLSTATE explizit auf '02000' (Last Record) und die ersten beiden Stellen des SQLSTATE auf ungleich '00', '01', '02' (Fehler) geprüft werden.

Da die SQLCA mit jedem SQL-Statement aktualisiert wird, sollte eine Prüfung auf SQLCODE oder SQLSTATE immer unmittelbar nach dem SQL-Befehl erfolgen. In einer DOW (Do While) oder DOU (Do Until)-Anweisung sollte eine Prüfung auf Unterfelder aus der SQLCA vermieden werden.

3.2.5.12.1 SQL-Befehl FETCH mit seriellem Cursor

Bei einem seriellen Cursor wird mit jedem FETCH die nächste Zeile aus dem Result-Set gelesen, bis das Ende des Result-Sets erreicht ist. Um den Cursor erneut an den Anfang zu positionieren, muss der Cursor zunächst geschlossen (SQL-Statement CLOSE) und anschließend wieder geöffnet (SQL-Statement OPEN) werden.

```
/FREE
    //Syntax
    Exec SQL  FETCH Next FROM MyCursor
              INTO :MyHostVar1, :MyHostVar2 :MyIndVar2,
                   :MyHostDS1,  :MyHostDS2  :MyIndArrDS2, ...;

    If SQLCODE = 100 or SQLSTATE = '02000';    //Last Row
       //Keine weiteren Zeilen/Sätze
    ElseIf     SQLCODE < *Zeros                //Fehler
            Or %Subst(SQLSTATE: 1: 2) > '02';
       //Fehler-Behandlung
    EndIf;
```

Syntax FETCH-Statement für seriellen Cursor

Im folgenden Beispiel werden alle Zeilen aus einem definierten und geöffneten Cursor in einer Schleife gelesen, die Daten werden in eine Host- mit zugehöriger Indikator-Struktur übertragen. Wenn das Ende der Daten erreicht (SQLSTATE '02000' oder SQLCODE 100) wird, kann man die Schleife verlassen. Im Fehlerfall (SQLCODE kleiner Null) wird die Schleife ebenfalls verlassen. Die Verarbeitung erfolgt für alle verbleibenden Zeilen.

3.2.5
Seite 92

```
/Free
  Exec SQL Declare CsrC1 Cursor For
           Select ...;

  Exec SQL Open  CsrC1;
  //Prüfung SQLCODE oder SQLSTATE und entsprechendes
    Fehlerhandling

  DoU 1=0;

      EXEC SQL Fetch Next From CsrC1
                    into :DSUmsatz :IndArrUmsatz;

      If SQLSTATE = '02000';              //Last Row
         leave;
      ElseIf SQLCODE < *Zeros;            //Error
         //Fehlerbehandlung
         leave;
      EndIf;

      //Verarbeitung

  EndDo;
```

Verarbeitung Result-Set für seriellen Cursor mit SQL-Statement FETCH

3.2.5.12.2 SQL-Befehl FETCH mit Scroll Cursor

Auch bei der Verarbeitung eines Scroll Cursors wird per Default immer die nächste Zeile verarbeitet.

Es ist jedoch auch möglich, eines der folgenden Schlüsselworte zu verwenden, um den Cursor innerhalb des Result-Sets neu zu positionieren und anschließend die entsprechende Zeile einzulesen. Die Angabe der Schlüsselworte erfolgt unmittelbar nach FETCH.

- NEXT
  Die nächste Zeile des Result-Sets wird eingelesen (Unterlassungswert).
- PRIOR
  Die der aktuellen Cursor-Position vorgelagerte Zeile wird eingelesen.
- FIRST
  Die erste Zeile des Result-Sets wird eingelesen.
- LAST
  Die letzte Zeile des Result-Sets wird eingelesen.
- BEFORE
  Der Cursor wird vor die erste Zeile positioniert, es werden jedoch keine Daten eingelesen. Deshalb darf bei BEFORE keine Ausgabe in den Host-Variablen/Strukturen erfolgen.
- AFTER
  Der Cursor wird nach der letzten Zeile positioniert, es werden jedoch keine Daten eingelesen. Deshalb darf bei AFTER keine Ausgabe in Host-Variablen/Strukturen erfolgen.
- CURRENT
  Der Cursor wird nicht erneut positioniert, stattdessen wird die aktuelle Zeile erneut eingelesen.
- RELATIVE n
  Mit **Relative n**, kann der Cursor um n-Zeilen innerhalb des Result-Sets verschoben werden.
  n kann durch eine positive oder negative ganze Zahl ersetzt werden. Es ist jedoch auch möglich **n** durch eine ganzzahlige Variable zu ersetzten.

Anmerkung:

Innerhalb eines Result-Sets ist es nicht möglich, einen oder mehrere Schlüssel zu positionieren. Zur Positionierung über Schlüsselwerte, muss der aktuell geöffnete Cursor geschlossen und nach dem Setzen der in den WHERE-Bedingungen verwendeten Host-Variablen-Werten erneut geöffnet werden.

```
/Free
   //Syntax
   Exec SQL  FETCH Next (Prior/First/Last/Before/After/
   Current/Relative n)
             FROM MyCursor
             INTO :MyHostVar1, :MyHostVar2 :MyIndVar2,
                  :MyHostDS1,  :MyHostDS2  :MyIndArrDS2, ...;

    If SQLCODE = 100 or SQLSTATE = '02000';    //Last Row
       //Keine weiteren Zeilen/Sätze
    ElseIf     SQLCODE < *Zeros                //Fehler
            Or %Subst(SQLSTATE: 1: 2) > '02';
       //Fehlerbehandlung
    EndIf;
```

Syntax FETCH-Statement für Scroll Cursor

Das folgende Beispiel zeigt die Verarbeitung eines Scroll Cursors in einem Dialog-Programm. Die Fetch-Methode (FetchMethod) wird abhängig von Aktionen, die im Dialog erfolgen (z. B. Drücken der Funktionstasten F17=Anfang, F18=Ende oder Blättern), gesetzt. Abhängig vom Inhalt der Variable FetchMethod wird der Cursor positioniert und die nächste Zeile eingelesen. Die Verarbeitungsschleife wird nur verlassen, wenn kein Satz gefunden wurde – z. B. bei FetchMethod EXIST oder einer unbekannten FetchMethod.

- Bei **FetchMethod NEW** wird der Cursor zunächst geschlossen und anschließend erneut geöffnet. Die FetchMethod wird auf FIRST gesetzt.
- Bei **FetchMethod FIRST** wird die erste Zeile eingelesen. Bei einem leeren Result-Set wird die Verabeitungsschleife verlassen.
- Bei **FetchMethod NEXT** wird die nächste Zeile eingelesen. Sofern keine weitere Zeile vorhanden ist, wird die FetchMethod auf LAST gesetzt, wodurch beim nächsten Durchlauf der letzte Satz eingelesen wird.
- Bei **FetchMethod PRIOR** wird die vorgelagerte Zeile eingelesen. Sofern keine vorgelagerte Zeile vorhanden ist, wird die FetchMethod auf FIRST gesetzt. Dadurch wird beim nächsten Durchlauf der erste Satz eingelesen.
- Bei **FetchMethod LAST** wird die letzte Zeile eingelesen.

- Bei **FetchMethod NEXTPAGE** wird der Cursor um die in der Konstante RowsPerPage gespeicherte Anzahl an Zeilen vorwärts bewegt und die so ermittelte Zeile eingelesen. Sofern die gewünschte Zeile nicht vorhanden ist, wird FetchMethod auf LASTPAGE gesetzt, wodurch beim nächsten Durchlauf auf die erste Zeile der letzten Seite positioniert wird.
- Bei **FetchMethod LASTPAGE** wird der Cursor zunächst ans Ende des Result-Sets positioniert, ohne die letzte Zeile einzulesen. Anschließend wird der so ermittelte Cursor um die in der Konstante RowsPerPage hinterlegte Anzahl an Zeilen rückwärts bewegt. Die so ermittelte Zeile wird nun eingelesen. Sofern die gewünschte Zeile nicht vorhanden ist, wird FetchMethod auf FIRST gesetzt, wodurch beim nächsten Durchlauf die erste Zeile eingelesen wird.
- Bei **FetchMethod PREVPAGE** wird der Cursor um die in der Konstante RowsPerPage gespeicherte Anzahl an Zeilen rückwärts bewegt – und die so ermittelte Zeile eingelesen. Sofern die gewünschte Zeile nicht vorhanden ist, wird FetchMethod auf FIRST gesetzt, wodurch beim nächsten Durchlauf die erste Zeile eingelesen wird.

```
/Free
   Exec SQL Declare CsrC1 Scroll Cursor For
                Select ... ;

   DoU 1=0;

       If Not isOpen;
          EXEC SQL Close CsrC1;
          EXEC SQL Open CsrC1;
          If SQLCODE < *Zeros;
             //Fehlerbehandlung
             Leave;
          EndIf;
          isOpen = *On;
          FetchMethod = 'FIRST';
       EndIf;

           //Einlesen der Zeilen abh. von der Fetch-Methode
       //-----------------------------------------------*
       Select;
       When FetchMethod = 'NEW';
            isOpen = *Off;
            Iter;
```

3.2.5
Seite 96

```
      When FetchMethod = 'FIRST';
            EXEC SQL Fetch First From CsrC1 into :DSUmsatz :IndArrUmsatz;
            If SQLCODE = 100;
               Leave;
            EndIf;

       When FetchMethod = 'NEXT' or FetchMethod = *BLANKS;
            EXEC SQL Fetch Next From CsrC1 into :DSUmsatz :IndArrUmsatz;
            If SQLCODE = 100;
               FetchMethod = 'LAST';
               Iter;
            EndIf;

       When FetchMethod = 'PRIOR';
            EXEC SQL Fetch Prior From CsrC1 into :DSUmsatz :IndArrUmsatz;
            If SQLCODE = 100;
               FetchMethod = 'FIRST';
               iter;
            EndIf;

       When FetchMethod = 'LAST';
            Exec SQL Fetch Last From CsrC1 into :DSUmsatz :IndArrUmsatz;

       When FetchMethod = 'NEXTPAGE';
            RelativeRows = RowsPerPage;
            Exec SQL Fetch Relative :RelativeRows From CsrC1
                     into :DSUmsatz :IndArrUmsatz;
            If SQLCODE = 100;
               FetchMethod = 'LASTPAGE';
               Iter;
            EndIf;

       When FetchMethod = 'LASTPAGE' or FetchMethod = 'PREVPAGE';
            RelativeRows = - RowsPerPage;
            If FetchMethod = 'LASTPAGE';
               EXEC SQL Fetch After From CsrC1;
            EndIf;
            Exec SQL Fetch Relative :RelativeRows From CsrC1
                     into :DSUmsatz :IndArrUmsatz;
            If SQLCODE = 100;
               FetchMethod = 'FIRST';
               Iter;
            EndIf;
```

```
      When FetchMethod = 'EXIT';
           Leave;
      Other;
         Leave;
      EndSL;

      If SQLCODE < *Zeros;            //Fehler
         Dsply 'Error';
         //Fehlerbehandlung
        leave;
      EndIf;

      //Verarbeitung inklusive Setzen der Fetch-Methode
      FetchMethod = 'NEXT';

   EndDo;
   ...
/End-Free
*********** Quelle EMBCSRSS2
```

Beispiel Fetch-Anweisungen bei einem Scroll Cursor (Quelle EmbCSRSS2)

3.2.5.13 SQL-Befehl CLOSE Cursor

Nach der Verarbeitung der Zeilen eines Result-Sets sollten sowohl ein serieller als auch ein Scroll Cursor explizit geschlossen werden.

Ein Cursor wird mit Hilfe des SQL-Statements CLOSE geschlossen.

Solange ein serieller Cursor geöffnet ist, kann er nicht neu positioniert werden. Solange der serielle und der sequentielle Cursor geöffnet sind, kann kein anderes Result Set – basierend auf geänderten Host-Variablen-Werten – empfangen werden.

Tipp:

Unmittelbar vor dem OPEN-Befehl den CLOSE-Befehl ausführen. War der Cursor geöffnet, so ist er jetzt geschlossen. War der Cursor nicht geöffnet, so kann der SQLCODE (-511) oder SQLSTATE (24501) einfach ignoriert werden.

```
/Free
    //Syntax
    Exec SQL  Close MyCursor;
```

Syntax CLOSE Cursor

3.2.5.14 Statisches SQL mit Cursor-Verarbeitung – ein Beispiel

Die Schritte, die zur Cursor-Verarbeitung notwendig sind und die bislang einzeln erklärt wurden, werden nun zusammengesetzt und in einer Prozedur ausgeführt.

Das folgende Beispiel zeigt eine Prozedur, in der abhängig von den übergebenen Parametern entweder die Umsätze für alle Kunden oder für einen einzelnen Kunden und ein bestimmtes Jahr ermittelt und mit Hilfe des OpCodes DSPLY angezeigt werden. Zusätzlich zu den kumulierten Kundenumsätzen wird auch der Gesamtumsatz für das angegebene Jahr und alle ausgewählten Kunden ermittelt.

Basierend auf den übergebenen Parameterwerten werden die Host-Variablen (HostVonDatum, HostBisDatum und HostKunde) bestückt. Die Host-Variable HostVonDatum wird mit dem 01. Januar des übergebenen Jahres versorgt, während die Host-Variable HostBisDatum mit dem 31. Dezember des gleichen Jahres versorgt wird. In die Variable HostKunde wird die Kundennummer übernommen, sofern der entsprechende Parameter übergeben wurde,.

Über die CASE-Anweisung in den WHERE-Bedingungen des DECLARE-CURSOR-Befehls wird gesteuert, ob nur die Umsätze für einen einzelnen Kunden oder für alle Kunden verarbeitet werden. Die CUBE-Klausel in der GROUP BY-Anweisung bewirkt, dass nicht nur die Umsätze auf Kundenebene kumuliert werden, sondern auch die Gesamtsumme ausgegeben wird.

Bevor der Cursor geöffnet wird, wird er zunächst geschlossen. Wenn der Cursor bei einer früheren Ausführung nicht geschlossen worden ist (z. B. das Programm wurde in einer Fehlersituation direkt verlassen), so ist er jetzt geschlossen und kann erneut positioniert werden. Durch das Öffnen werden die Daten aktualisiert und der Cursor vor die erste Zeile des Result-Sets positioniert.

Da nur ein serieller Cursor definiert wurde, kann immer nur die nächste Zeile im Result-Set verarbeitet werden. In einer Verarbeitungsschleife wird jeweils über eine FETCH-Anweisung die nächste Zeile in eine Host-Struktur ausgegeben. Die Angabe einer zusätzlichen Indikator-Feldgruppe ermöglicht die Prüfung auf NULL-Werte, die im Result-Set ausgegeben wurden. Unmittelbar nach dem FETCH-Statement wird geprüft, ob das Ende des Result-Sets erreicht worden oder ob ein Fehler aufgetreten ist. In beiden Fällen wird die Verarbeitungsschleife verlassen.

Die eigentliche Verarbeitung in diesem Beispiel besteht lediglich darin, dass ein Text mit Kundennummer und Kundenumsatz aufbereitet worden ist, der mit Hilfe des OpCodes DSPLY angezeigt wird.

Nachdem die Verarbeitungsschleife verlassen wurde, wird der Cursor geschlossen und die Prozedur beendet.

3.2.5
Seite 100

```
P UmsKundeJahr    B                   Export

D UmsKundeJahr    PI
D   ParJahr                      4P 0                   Const
D   ParKundeNr                  15A   Options(*NoPass)  Const

D HostKunde       S              15A
D HostVonDatum    S                D   DatFmt(*ISO)
D HostBisDatum    S                D   DatFmt(*ISO)

D DSUmsatz        DS                   Qualified
D   Kunde                        15A
D   Total                        11S 0

D IndArrUmsatz    S               5I 0 Dim(2)
 //-----------------------------------------------------------------
 /Free
    If %Parms   >= %ParmNum(ParKundeNr);
       HostKunde = %Trim(ParKundeNr);
    EndIf;

    HostVonDatum = %Date(%EditC(ParJahr: 'X') + '-01-01': *ISO);
    HostBisDatum = %Date(%EditC(ParJahr: 'X') + '-12-31': *ISO);

    Exec SQL Declare CsrC1 Cursor For
                Select KundeNr, Sum(Wert)
                   from Umsatz
                   where     VerkDatum between :HostVonDatum
                                           and :HostBisDatum
                         and KundeNr = Case When :HostKunde > ' '
                                            Then :HostKunde
                                            Else KundeNr End
                Group By Cube(KundeNr)
             For Read Only  Optimize For 10 Rows;

    Exec SQL Close CsrC1;
    Exec SQL Open  CsrC1;
    If SQLCODE < *Zeros;
       //Fehlerbehandlung;
       Return;
    EndIf;
```

```
     DoU 1=0;

         EXEC SQL Fetch Next From CsrC1 into :DSUmsatz :IndArrUmsatz;
         If SQLSTATE = '02000';              //Last Row
           Leave;
         ElseIf SQLCODE < *Zeros;            //Error
           //Fehlerbehandlung
           Leave;
        EndIf;

        If IndArrUmsatz(1) < *Zeros;    //NULL Value KundeNr = GesamtSumme
           Dsply ('Gesamt: ' + %Char(DSUmsatz.Total));
        Else;
           Dsply ('Kunde: ' + %Trim(DSUmsatz.Kunde) + ' ' +
                              %Char(DSUmsatz.Total));
        EndIf;

     EndDo;

     Exec SQL Close CsrC1;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung
     EndIf;

     Return;
  /End-Free
 P UmsKundeJahr...
 P                 E
  ********* Quelle EMBCSRSS1
```

Verarbeitung eines Cursors in statischem SQL – Beispiel

3.2.5.15 Fortschreiben von Datensätzen bei Cursor-Verarbeitung

Bislang wurden über einen Cursor lediglich Datensätze eingelesen. Die eingelesenen Daten wurden jedoch weder modifiziert, noch zurückgeschrieben, noch gelöscht.

Die eingelesenen Datensätze können zwar über ein separates UPDATE- oder DELETE-Statement, das nicht mit dem Cursor in Verbindung steht, modifiziert oder gelöscht werden. Voraussetzung ist aber, dass der eindeutige Schlüssel des Datensatzes oder die relative Satznummer des Datensatzes bekannt ist.

Die Verwendung eines separaten UPDATE oder DELETE-Statements hat zudem den Nachteil, dass für die Ausführung dieser SQL-Befehle jeweils ein eigener Datenpfad erstellt werden muss – d. h., es muss eine komplette Optimierung ausgeführt werden (FULL OPEN). Sofern kein eindeutiger Schlüssel ermittelt werden konnte und ein Update oder Delete über die relative Satznummer erfolgen muss, führt dies zu einer schlechten Performance. Im Gegensatz zu native I/O, bei dem der Zugriff über die relative Satznummer die beste Zugriffsmethode ist, muss bei SQL in Verbindung mit der relativen Satznummer ein Table-Scan (oder Table-Probe) ausgeführt werden – d. h., alle Datensätze der Tabelle werden nach der gewünschten relativen Satznummer durchsucht.

Es besteht jedoch die Möglichkeit, einen Datensatz über die Cursor-Verarbeitung zu modifizieren oder zu löschen. Hierzu muss der Cursor „deletable“ (löschbar) oder „updatable“ (änderbar) sein – d. h., das in der DECLARE-CURSOR-Anweisung hinterlegte SELECT-Statement muss folgende Voraussetzungen erfüllen:

- Im SELECT-Statement darf nur eine einzige Tabelle oder View angegeben werden.
  JOIN-Anweisungen, Common-Table-Expressions (CTE) oder verschachtelte Sub-Selects sind nicht erlaubt.
  Wenn eine View angegeben wurde, so darf sich diese View ebenfalls nur auf eine einzige Tabelle/Datei beziehen. Sofern für Views, in denen Tabellen/Dateien verknüpft wurden INSTEAD OF TRIGGER für DELETE oder UPDATE generiert wurden, können auch Datensätze aus solchen Views geändert oder gelöscht werden.
- GROUP BY und HAVING-Anweisungen, Aggregat-Funktionen sowie die Angabe von DISTINCT sind nicht erlaubt.
- UNION, INTERSECT oder EXCEPT-Anweisungen sind nicht zulässig.

- Sofern im SQL-Statement eine ORDER BY-Anweisung verwendet wird, ist die Angabe von FOR UPDATE mit oder ohne Spaltenliste erforderlich.
- Sofern die UPDATEs oder DELETEs für einen Scroll Cursor erfolgen sollen, muss DYNAMIC SCROLL angegeben werden.

Die Verarbeitung von änder- oder löschbaren Cursors entspricht der Verarbeitung von Cursors, bei denen nur Daten gelesen werden – d. h., das OPEN-, FETCH- und CLOSE-Statement sind identisch. Die einzige Ausnahme bilden multiple-Row-Fetches, die in Verbindung mit Update oder Delete nicht erlaubt sind.

Um Datensätze über einen Cursor zu modifizieren oder zu löschen muss beim DELETE- oder UPDATE-Statement die Erweiterung WHERE CURRENT OF Cursor angegeben werden.0

```
/Free
   //Syntax: DELETE Zeile über Cursor
   Delete From MyTable
      WHERE CURRENT OF MyCursor;

   //Syntax: UPDATE Zeile über Cursor
   Update MyTable
      Set MyColumn1 = :Wert1, MyColumn2 = :Wert2, ...
          MyColumnN = :WertN
      WHERE CURRENT OF MyCursor;
```

Syntax UPDATE und DELETE bei Cursor-Verarbeitung

Achtung:

Sofern die Erweiterung WHERE CURRENT OF fehlt, werden alle Datensätze in der angegebenen Tabelle geändert oder gelöscht!

Sofern eine DELETE- oder UPDATE-Anweisung für einen Cursor mit WHERE CURRENT OF definiert wurde, wird die Zeile beim FETCH gesperrt.

Die Freigabe des Datensatzes erfolgt, sofern ohne Commitment-Steuerung gearbeitet wurde, entweder nach Ausführung des Update- oder Delete-Statements bzw. sobald die nächste Zeile gelesen wird. Wenn mit Commitment-Steuerung gearbeitet wird, erfolgt die Freigabe erst bei Ausführung eines COMMIT- oder ROLLBACK-Befehls. Dabei spielt es keine Rolle, in welcher (Programmier-)Sprache der COMMIT- oder ROLLBACK- (bzw. **Rolbk-**) Befehl abgesetzt wurde.

3.2.5
Seite 104

Das folgende Beispiel zeigt die Prozedur **ChgPreisAll**, die ohne Parameter aufgerufen wird. In dieser Prozedur werden alle im Artikelstamm hinterlegten Artikelpreise abhängig von der Artikelgruppe (Stelle 1-2 der Artikel-Nr.) um einen bestimmten Prozentsatz verändert.

Die Artikelgruppen sowie die Prozentsätze, um die die Preise erhöht werden sollen, sind in einer globalen Datenstruktur hinterlegt. Die neuen Preise werden innerhalb der Prozedur – basierend auf diesen Informationen – berechnet.

Im DECLARE-CURSOR-Statement wird durch die Angabe von FOR UPDATE OF PREIS festgelegt, dass nur die Spalte **Preis** aus dem Artikelstamm **ArtikelStX** geändert werden darf. Da keine WHERE-Anweisung angegeben wurde, werden alle Artikel ohne Einschränkung verarbeitet.

Der Cursor wird zunächst geschlossen und anschließend geöffnet. Sofern beim Öffnen des Cursors ein Fehler auftritt, wird die Prozedur beendet.

Über eine Schleife wird eine Zeile nach der anderen in die Host-Struktur **DSCsrC1** mit den Unterfeldern **Artikel** und **Preis** eingelesen. Unmittelbar nach der FETCH-Anweisung wird geprüft, ob das Ende des Result-Sets erreicht worden (SQLCODE = 100) oder ob ein Fehler aufgetreten ist (SQLCODE < *Zeros). In beiden Fällen wird die Verarbeitungsschleife verlassen.

Der neue Preis wird basierend auf dem eingelesenen alten Preis, der Artikelgruppe und dem in der globalen Datenstruktur hinterlegten Prozentsatz berechnet. Sollte bei der Berechnung ein Fehler auftreten, wird dieser über eine Monitor-Group abgefangen. Im Fehlerfall erfolgt kein Update, stattdessen wird die nächste Zeile eingelesen. Konnte ein neuer Preis errechnet werden, wird der Datensatz über eine UPDATE-Anweisung mit der Erweiterung WHERE CURRENT OF fortgeschrieben.

Konnte der neue Preis ordnungsgemäß fortgeschrieben werden, werden der Artikel, der alte und der neue Preis mit Hilfe des OpCodes DSPLY angezeigt.

```
* Tabelle: Artikel-Gruppe und Preisänderung in Prozent/Artikel-Gruppe
D DSChange        DS                   Qualified
D   ArtGrp1                     2A    inz('BW')
D   Proz1                       5P 2  inz(5)
D   ArtGrp2                     2A    inz('JG')
D   Proz2                       5P 2  inz(3)
D   ArtGrp3                     2A    inz('SK')
D   Proz3                       5P 2  inz(10)
D ArrAll                              Dim(3) Overlay(DSChange)
D  ArrArtGrp                    2A    Overlay(ArrAll)
D  ArrProz                      5P 2  Overlay(ArrAll: *Next)
 ******************************************************************************
P ChgPreisAll     B                   Export

D ChgPreisAll     PI

D DSCsrC1         DS                  Qualified Inz
D   Artikel                    22A
D   Preis                      11P 2

D ArtGrp          S              2A
D PreisNeu        S                   like(DSCsrC1.Preis)
D DsplyText       S             50A
 *---------------------------------------------------------------------
 /Free
    Exec SQL Declare CsrC1 Cursor For
                Select  ArtikelNr, Preis
                   From ArtikelStX
                For Update Of Preis;

     EXEC SQL Close CsrC1;
     EXEC SQL Open  CsrC1;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung;
        Return;
     EndIf;

     DoU 1 = 0;
         Exec SQL Fetch Next From CsrC1 into :DSCsrC1;
         If SQLCODE = 100;
            Leave;
         ElseIf SQLCODE < *Zeros;
            Dsply SQLCODE;
            Leave;
         EndIf;
```

3.2.5
Seite 106

```
          Monitor;
             ArtGrp   = DSCsrC1.Artikel;
             PreisNeu = DSCsrC1.Preis *
                        (1 + DSChange.ArrProz(%LookUp(ArtGrp:
                                              DSChange.ArrArtGrp)) / 100);
          On-Error;
             iter;
          EndMon;

          EXEC SQL  Update ArtikelStX
                       Set Preis = :PreisNeu
                     Where Current Of CsrC1;
          If SQLCODE < *Zeros;
             Dsply SQLCODE;
             iter;
          EndIf;

          DsplyText =  'Artikel: '    + %Trim(DSCsrC1.Artikel) +
                       ' Preis alt: ' + %Char(DSCsrC1.Preis)   +
                       ' Preis neu: ' + %Char(PreisNeu);
          Dsply DsplyText;

       EndDo;

       EXEC SQL Close CsrC1;
       Return;

 /End-Free
P ChgPreisAll     E
 ************* Quelle EMBCSRSU1
```

Beispiel Ändern Datensätze/Zeilen mit Cursor-Verarbeitung

3.2.5.16 Multiple Row Fetches – Einlesen von Datenblöcken

Im Gegensatz zu native I/O, bei dem immer nur ein Datensatz nach dem anderen verarbeitet werden kann, können bei der Cursor-Verarbeitung mehrere Zeilen (maximal 32.767) gleichzeitig in eine Mehrfachdatenstruktur oder eine Array-Datenstruktur eingelesen werden.

Nach dem erfolgreichen (Multiple-Row) FETCH wird der Cursor auf der zuletzt eingelesenen Zeile positioniert, so dass beim nächsten (Multiple-Row) FETCH der folgende Datenblock korrekt eingelesen werden kann.

Die DECLARE, OPEN und CLOSE-Statements sind für Single- und Multiple-Row-Fetches identisch. So ist es auch möglich, Scroll Cursor für Multiple-Row-Fetches zu definieren.

Die Unterscheidung zwischen Single- und Multiple-Row-Fetches erfolgt im FETCH-Statement. Bei einem Multiple-Row-Fetch muss zum einen die Anzahl der in einem Block einzulesenden Zeilen angegeben und zum anderen für die Ausgabe eine Mehrfach- oder Array-Datenstruktur verwendet werden. Die Anzahl der einzulesenden Zeilen kann entweder als ganzzahliger Wert oder ganzzahlige Host-Variable angegeben werden.

```
/Free
  //Syntax: FETCH for Multiple Row Fetches
  Exec SQL Fetch Next From MyCursor
                 For x Rows into :HostStructure
                    :HostIndStructure;
```

Syntax FETCH-Statement for Multiple-Row-Fetches

Wenn mehrere Zeilen eingelesen werden, so ist es wichtig zu wissen, wie viele Zeilen tatsächlich übertragen wurden und ob noch weitere Zeilen vorhanden sind. Diese Informationen werden in der SQLCA ausgegeben. Sie können aber auch über die Diagnostics Area abgefragt werden:

- SQLERRD(3)/SQLER3
  Die Anzahl der mit einem Fetch eingelesenen Zeilen entspricht dem Schlüsselwort ROW_COUNT in der Diagnostics Area.
- SQLERRD(5)/SQLER5
  Sofern die letzte Zeile des Result-Sets eingelesen wurde, wird in diesem Datenstruktur-Unterfeld +100 ausgegeben. Das entspricht dem Schlüsselwort DB2_LAST_ROW in der Diagnostics Area.

3.2.5
Seite 108

Das folgende Beispiel zeigt eine Prozedur, die im Zusammenhang mit dem seitenweisen Lesen – z. B. beim seitenweisen Füllen von Subfiles oder Web-Sites – verwendet werden kann. Die Anzahl der Zeilen pro Seite sowie die zu lesende Seite werden als Parameterwerte übergeben.

Durch die OLAP-Ranking-Funktion ROW_NUMBER() werden die Zeilen im Result-Set fortlaufend nummeriert. Die Seitennummer wird mit Hilfe der skalaren Funktion CEILING und der Anzahl der übergebenen Zeilen berechnet und mit der als Parameterwert übergebenen gewünschten Seite verglichen.

Die angeforderten Zeilen für die gewünschte „Seite“ werden in einem einzigen FETCH-Statement in eine Array-Datenstruktur übertragen. Diese Array-Datenstruktur wird in Form eines Ausgabeparameters an die rufende Prozedur zurückgegeben.

Die Anzahl der in das Result-Set übertragenen Zeilen (SQLER3/SQLERRD(3)) wird als Rückgabewert an die rufende Prozedur ausgegeben.

```
D RefDSUmsatz     DS                  Qualified Inz Template
D   Kunde                       15A
D   Artikel                     22A
D   Datum                         D    DatFmt(*ISO)
D   Wert                        11P 2

D RefElem         C                   Const(50)
 ******************************************************************************
P UmsatzPage      B                   Export

D UmsatzPage      PI             5I 0
D   POutDSUmsatz                      Like(RefDSUmsatz) Dim(RefElem)
D   ParPage                     5I 0                   Const
D   ParRowsPage                5I 0                   Const
D   ParJahr                    4P 0                   Const

D RtnNbrRows      S              5I 0

D HostVonDatum    S               D   DatFmt(*ISO)
D HostBisDatum    S               D   DatFmt(*ISO)

D DSUmsatz        DS                  Qualified Dim(50) Inz
D   Kunde                       15A
D   Artikel                     22A
D   Datum                         D   DatFmt(*ISO)
D   Wert                        11P 2
 //-----------------------------------------------------------------------
 /Free
```

```
     Clear POutDSUmsatz;
     If ParPage <= *Zeros or ParRowsPage <= *Zeros or ParJahr < 2000;
        Return -1;
     EndIf;

     HostVonDatum = %Date(%EditC(ParJahr: 'X') + '-01-01': *ISO);
     HostBisDatum = %Date(%EditC(ParJahr: 'X') + '-12-31': *ISO);

     Exec SQL
          Declare CsrC1 Cursor For
             with x as (Select RowNumber() Over(Order By KundeNr,
                                                ArtikelNr, VerkDatum) xRow,
                                KundeNr, ArtikelNr, VerkDatum, Wert
                           from Umsatz
                           Where Verkdatum between :HostVonDatum
                                                and :HostBisDatum)
             Select KundeNr, ArtikelNr, VerkDatum, Wert
                from x
                Where Ceil(xRow / Dec(:ParRowsPage, 11, 2)) = :ParPage
                Order By xRow;

     Exec SQL Close CsrC1;
     Exec SQL Open  CsrC1;
     If SQLCODE < *Zeros;
        Return SQLCODE;
     EndIf;

     Exec SQL Fetch Next From CsrC1 For :ParRowsPage Rows into :DSUmsatz;
     If SQLCODE < *Zeros;
        RtnNbrRows = SQLCODE;
     Else;
        POutDSUmsatz = DSUmsatz;
        RtnNbrRows = SQLERRD(3);
     EndIf;

     Exec SQL Close CsrC1;
     Return RtnNbrRows;

  /End-Free
 P UmsatzPage      E
  ************** Quelle EMBCSRSM1
```

Beispiel Cursor-Verarbeitung – Multiple-Row-Fetch (Quelle EmbCsrSM1)

Performancebetrachtungen zu Multiple-Row-Fetches

Durch die Verwendung von Multiple-Row-Fetches kann man, anstatt die Zeilen einzeln einzulesen, die Anzahl der Datenbankzugriffe beträchtlich reduzieren, damit die Performance verbessert wird.

In der folgenden Tabelle werden Single- und Multiple-Row-Fetches mit geblocktem und ungeblocktem Lesen verglichen.

	Einlesen von 100 Zeilen	
	Database Manager ohne Blocken	**Database Manager mit Blocken**
Single Row Fetch (100 einzelne Fetch-Anweisungen)	100 SQL Calls 100 Datenbank-Zugriffe	100 SQL Calls 1 Datenbank-Zugriff
Multiple Row Fetch (1einzelne Fetch-Anweisung)	1 SQL-Run time Call 100 Datenbank-Zugriffe	1 SQL-Run time Call 1 Datenbank-Zugriff

Gegenüberstellung Single- und Multiple-Row-Fetches

Bei der Verwendung von Multiple-Row-Fetches sollte im Hinblick auf die Performance Folgendes beachtet werden:

- Bei einem Multiple Row-Fetch sollten **so viele Zeilen wie möglich** in einem FETCH eingelesen werden. Das Result-Set wird bereits bei Ausführung des OPEN-Statements erstellt, was der zeitaufwendigste Prozess bei der Ausführung von embedded SQL-Befehlen mit Cursor-Verarbeitung darstellt. Beim FETCH werden nur noch die Zeilen aus dem Result-Set abgerufen. Die Blockung der Daten wird vom Datenbank-Manager übernommen.
  Mit einem einzigen FETCH-Statement können bis zu 32.767 Zeilen empfangen werden. Die maximale Anzahl der Zeilen, die empfangen werden können, hängt zum einen vom Umfang der übertragenen Informationen (Spalten, Datentypen und Länge) und zum anderen von der maximal zulässigen Größe von Mehrfach- bzw. Array-Datenstrukturen ab.
  Je gezielter die Daten ausgewählt werden (z. B. anstatt des kompletten Datensatzes (SELECT *) werden nur die fünf benötigten Spalten ausgewählt), desto **weniger Datenbankoperationen** sind notwendig.
- Beim **Wechsel von Multiple- zu Single-Row-Fetches** (und umgekehrt) innerhalb des gleichen geöffneten Cursors ist eine Blockung der Datensätze nur eingeschränkt möglich – d. h., intern müssen die Zeilen trotz des Multiple-Row-Fetches einzeln ausgegeben werden.
- Werden Scroll Cursor in Verbindung mit multiple Row-Fetches verwendet, sollte die Verarbeitung der Zeilen über **CURRENT, PRIOR und RELATIVE vermieden** werden, da auch bei diesen Aktionen eine geblockte Verarbeitung nicht möglich ist.

3.2.5.17 Dynamisches SQL mit Cursor

Bei statischem SQL ist der SQL-Befehl bereits zur Kompilierungszeit bekannt. Bei dynamischem SQL wird der SQL-Befehl erst zur Laufzeit gebildet. Sofern mit einem Cursor gearbeitet wird, bedeutet dynamisches SQL, dass das SELECT-Statement, das im DECLARE CURSOR-Statement hinterlegt wird, zur Laufzeit aufbereitet wird. Bei der Cursor-Verarbeitung muss zudem berücksichtigt werden, dass die Ergebniszeilen in Host-Variablen oder Host-Strukturen ausgegeben werden müssen, um weiterverarbeitet werden zu können.

Werden SELECT-Statements in Verbindung mit der Cursor-Verarbeitung dynamisch aufbereitet, muss man zwischen den folgenden Methoden unterscheiden:

Dynamisches SQL mit fixer Auswahlliste
Host-Strukturen und Host-Variablen können in dynamischem SQL mit Cursor-Verarbeitung nur dann verwendet werden, wenn innerhalb des SELECT-Statements immer die gleichen Spalten/Felder in der gleichen Reihenfolge ausgewählt werden.

Dynamische SELECT-Statements unterscheiden sich durch den Zugriff auf unterschiedliche Tabellen/Dateien/Views in unterschiedlichen Bibliotheken – jedoch mit den gleichen Spalten/Feld-Namen – sowie durch unterschiedliche WHERE-Bedingungen oder unterschiedliche Sortierungen.

Dynamisches SQL mit variabler Auswahlliste
Bei dynamischen SELECT-Statements kann jede beliebige Spalte/Feld aus jeder beliebigen Tabelle/Datei/View ausgewählt werden. Es ist außerdem möglich, beliebige neue Spalten/Felder zu generieren (z. B. kann in einer zusätzlichen Spalte der Gesamtwert über Menge * Preis ermittelt werden).

Da zur Kompilierungszeit weder Informationen über die ausgewählten Tabellen, die ausgewählten Spalten oder die zusätzlich generierten Spalten vorliegen, können weder Host-Strukturen noch Host-Variablen angelegt und verwendet werden.

Um die im Result-Set ausgegebenen Daten dennoch empfangen und verarbeiten zu können, muss mit einer SQL Descriptor Area gearbeitet werden. In der **SQL Descriptor Area** werden Pointer/Adressen der ausgegebenen Daten verwaltet. Ebenso enthält die SQL Descriptor Area Informationen über den Datentyp sowie die Länge und Anzahl Nachkommastellen der Daten.

In den meisten Fällen, in denen dynamisches SQL in Verbindung mit einem SELECT-Statement eingesetzt wird, kann jedoch eine fixe Auswahlliste verwendet werden.

Ergänzung 21/2012

In Dialogprogrammen oder einfachen Listengeneratoren, in denen die Anwender auswählen können, welche Informationen angezeigt, aufbereitet oder ausgedruckt werden sollen, sind zumindest die Tabellen/Dateien/Views und damit die potenziell auswählbaren Spalten/Felder bekannt. Damit kann z. B. der komplette Datensatz in einer Host-Struktur empfangen werden, und nur die benötigten Informationen werden anschließend angezeigt.

In nur ganz wenigen Ausnahmefällen ist eine volldynamische Verarbeitung erforderlich.

Zur Verarbeitung von dynamischem SQL in Verbindung mit einem Cursor sind die folgenden Schritte erforderlich:

Aufbereitung des SQL SELECT-Statements (Statement String)
Das SELECT-Statement muss zur Laufzeit zunächst als Text-String (Statement String) aufbereitet werden.

Werden im Text variable Werte verwendet, so müssen diese entweder im Text aufgelöst werden oder können – sofern es sich nicht um Tabellen/Dateien/Views oder Bibliotheken handelt – als Parameter-Marker integriert werden.

Werden die Variablen aufgelöst, so müssen alphanumerische Werte und Datumswerte in Hochkommata eingeschlossen werden. Einfache Hochkommata müssen dabei verdoppelt werden. Numerische Werte werden nicht in Hochkommata eingeschlossen, müssen jedoch, um eingebunden werden zu können, mit Hilfe der Built-In-Function %CHAR() konvertiert werden.

Datumswerte sollten in einem Datumsformat mit einer vierstelligen Jahresangabe (*ISO, *EUR, *USA oder *JIS) aufbereitet werden.

Sofern Unicode- (Datentyp C) oder andere DBCS-Werte (Datentyp G) eingebunden werden sollen, sollte der SQL-String als Unicode (Datentyp C) oder grafisch (Datentyp G) aufbereitet werden.

Werden Parameter-Marker verwendet, wird anstelle der variablen Werte ein Fragezeichen (?) angegeben. Im Gegensatz zum direkten Einbinden der variablen Werte innerhalb des Strings, muss bei Verwendung von Parameter-Markern keine Rücksicht auf den Datentyp der variablen Werte genommen werden, d. h. weder eine Einbettung in Hochkommata, noch eine Konvertierung in Unicode ist erforderlich.

Konvertierung in ausführbares SQL-Statement (Prepared Statement)
Der aufbereitete Text (Statement String) muss in ein ausführbares SQL-Statement (Prepared Statement) konvertiert werden. Dabei spielt es zunächst keine Rolle, ob der Text Parameter-Marker enthält oder nicht.

Bei der Konvertierung in ein ausführbares SQL-Statement wird zunächst geprüft, ob es sich um ein gültiges SQL-Statement handelt (Syntaxprüfung). Sofern das SQL-Statement fehlerhaft ist, werden ein entsprechender SQLCODE sowie SQLSTATE in die SQLCA (1.3.3.3 SQL Communication Area (SQLCA)) ausgegeben. Da im Gegensatz zu reinen RPG-Programmen im Fehlerfall kein Programmabbruch erfolgt, sollte nach der Ausführung eines jeden SQL-Statements entweder der SQLCODE oder der SQLSTATE geprüft werden und im Fehlerfall eine entsprechende Fehlerbehandlung eingeleitet werden, z. B. die Anzeige des Nachrichtentextes.

Die Konvertierung des aufbereiteten SELECT-Statements erfolgt mit Hilfe des SQL-Befehls PREPARE.

Zuweisung des konvertierten SELECT-Statements zu einem Cursor
Im Gegensatz zu dynamischem SQL ohne Cursor, bei dem das konvertierte SQL-Statement direkt mit dem SQL-Befehl EXECUTE ausgeführt wird, muss bei der Verarbeitung über einen Cursor das Prepared Statement zunächst einem Cursor zugeordnet werden.

Die Zuweisung erfolgt über das SQL-Statement DECLARE CURSOR.

Verarbeitung des Cursors mittels fixer Auswahlliste
Sofern mit einer fixen Auswahlliste gearbeitet wird, unterscheidet sich die weitere Verarbeitung nicht von der Cursor-Verarbeitung mit statischem SQL.

- **OPEN CURSOR**
  Der deklarierte Cursor muss geöffnet werden.
  Die Verwendung von Host-Variablen innerhalb eines dynamischen Select-Statements ist nicht zulässig. Dennoch können variable Werte in Form von Parameter-Markern (?) im Statement-String hinterlegt werden.
  Durch die Verwendung von Parameter-Markern kann das gleiche SQL-Statement einmalig mit Hilfe des SQL-Statements PREPARE konvertiert, jedoch mehrfach mit unterschiedlichen Werten ausgeführt werden.
  Bevor der Cursor erneut geöffnet werden kann, muss er zunächst geschlossen werden. Nur ein geschlossener Cursor kann geöffnet werden.
  Sofern mit Parameter-Markern gearbeitet wurde, müssen die variablen Werte beim OPEN-Statement angegeben werden.

3.2.5
Seite 114

- **FETCH CURSOR**
  Über den SQL-Befehl FETCH werden die ermittelten Daten aus dem Result-Set in Host-Variablen oder Host-Strukturen übertragen.
  Mit Hilfe von Indikator-Variablen oder Indikator-Strukturen kann auf ausgegebene NULL-Werte geprüft werden.
  Da auch bei dynamischem SQL Scroll-Cursor definiert werden können, ist es zudem möglich, den Cursor in Verbindung mit den Schlüsselwörtern NEXT/FIRST/LAST/PRIOR/BEFORE/AFTER oder RELATIVE n innerhalb des Result-Sets vor- oder rückwärts zu bewegen oder zu positionieren.
  Die Ausgabe von mehreren Zeilen in Mehrfach- oder Array-Datenstrukturen (Multiple Row Fetches) ist auch in Verbindung mit dynamischem SQL zulässig.
- **UPDATE/DELETE mit WHERE CURRENT OF**
  Sofern Datensätze über einen dynamischen Cursor geändert oder gelöscht werden sollen, kann dies wie bei der statischen Cursor-Verarbeitung über ein UPDATE- oder DELETE-Statement mit der Erweiterung WHERE CURRENT OF erfolgen.
- **CLOSE CURSOR**
  Der Cursor muss geschlossen werden.
  Sofern der Cursor nicht geschlossen wurde, ist es weder möglich dem Cursor ein anderes SQL-Statement zuzuordnen noch den deklarierten Cursor (mit veränderten Parameter-Marker-Werten) neu zu öffnen.
  Der Cursor muss also vor der (erneuten) Ausführung des PREPARE-Statements geschlossen sein. Anderenfalls kann weder das zugrundeliegende SQL Statement konvertiert, noch können die neuen Daten geladen werden, noch kann der (serielle) Cursor an den Anfang der Daten positioniert werden.
  Wird mit Parameter-Markern gearbeitet und das SQL-Statement einmalig konvertiert und mehrfach ausgeführt, muss der Cursor vor dem (erneuten) Öffnen geschlossen sein.

Definition und Verwendung der SQLDA bei dynamischem SQL mit variabler Auswahlliste

Da bei einer dynamischen Auswahlliste zur Kompilierungszeit weder bekannt ist, welche Spalten ausgewählt oder generiert werden, noch welchen Datentyp oder welche Länge die Spalten haben, können die Daten aus dem Result-Set nicht direkt in Host-Variablen oder Host-Strukturen übertragen werden. Stattdessen können diese Informationen in einer **SQL Descriptor Area** (**SQLDA**) bereitgestellt und aus ihr ausgelesen werden.

Die Definition und Verwendung einer SQL Descriptor Area ist nur für dynamische SELECT-Statements mit variabler Auswahlliste erforderlich. Bei einer fixen Auswahlliste sind die Spalten sowie deren Reihenfolge, Datentypen und Längen zur Kompilierungszeit bekannt. Damit können die über das Result-Set ausgegebenen Informationen direkt in vordefinierte Host-Variablen oder Host-Strukturen übertragen werden.

Die SQLDA muss über eine SQL **INCLUDE-Anweisung** am Ende der globalen oder lokalen D-Bestimmungen eingebunden werden.

Die Definition der SQLDA allein ist noch nicht ausreichend, da innerhalb einer Quelle mehrere dynamische SQL-Statements mit variabler Auswahlliste hinterlegt werden können. Aus diesem Grund muss die SQLDA dem dynamischen SELECT-Statement (Prepared Statement) bzw. dem Cursor zugeweisen werden. Die Zuweisung kann wie folgt vorgenommen werden:

- **DECRIBE**
  Die Zuordnung der SQL Descriptor Area kann über die Ausführung des SQL-Befehls DESCRIBE erfolgen.
- **PREPARE**
  Die SQL Descriptor Area kann auch bei der Konvertierung des Statement-Strings in ein ausführbares SQL-Statement über den SQL-Befehl PREPARE angegeben werden.

Bei der weiteren Verarbeitung müssen die gleichen Schritte ausgeführt werden wie bei der Cursor-Verarbeitung mit statischem SQL oder dynamischem SQL mit fixer Auswahlliste:

- **DECLARE CURSOR**
  Das konvertierte SQL-Statement muss einem Cursor zugeordnet werden.
- **OPEN CURSOR**
  Der zugeordnete Cursor wird geöffnet.
  Sofern im Statement-String Parameter-Marker verwendet wurden, werden die Host-Variablen, die die variablen Werte enthalten, an dieser Stelle angegeben.

- **FETCH CURSOR**
  Die Datensätze des Result-Sets werden einzeln oder en bloc (Multiple Row Fetch) eingelesen.
  Bei dynamischem SQL mit variabler Auswahlliste erfolgt die Ausgabe jedoch nicht in vordefinierte Host-Variablen oder Host-Strukturen, sondern in die SQL Descriptor Area.
  Die Verarbeitung der in die Descriptor Area eingelesenen Daten muss über Pointer-Handling erfolgen.
- **CLOSE CURSOR**
  Der Cursor muss geschlossen werden, da ansonsten weder ein neues SQL-Statement konvertiert noch der Cursor erneut (mit anderen Variablen-Werten) geöffnet und positioniert werden kann.

3.2.5.17.1 Dynamisches SQL mit fixer Auswahlliste

Bei der Cursor-Verarbeitung mit dynamischem SQL und fixer Auswahlliste können die meisten Schritte, die auch für die statische Cursor-Verarbeitung erforderlich sind, in der gleichen Form ausgeführt werden.

Für dynamisches SQL mit fixer Auswahlliste gibt es nur zwei Abweichungen gegenüber der statischen Cursor-Verarbeitung:

- **PREPARE**
  Das SELECT-Statement muss komplett als Text aufbereitet und mit Hilfe des SQL-Befehls PREPARE in ein ausführbares SQL-Statement (Prepared Statement) konvertiert werden.
- **DECLARE CURSOR**
  Das konvertierte SQL SELECT-Statement (Prepared Statement) muss im DECLARE CURSOR-Statement anstelle des SELECT-Statements angegeben werden.
- **Alle anderen Schritte**
  OPEN-, FETCH- und CLOSE-Statements können wie bei statischem SQL mit Cursor ausgeführt werden.

3.2.5.17.1.1 SQL-Befehl PREPARE mit dynamischen SELECT-Statements

Bei dynamischem SQL mit Cursor-Verarbeitung muss zunächst das komplette SELECT-Statement als Text aufbereitet werden (Statement String).

Dabei ist zu beachten, dass auch Anweisungen wie FOR UPDATE OF oder OPTIMIZE FOR x ROWS Bestandteil des SELECT-Statements sind und demnach – sofern erforderlich – in den String integriert werden müssen.

Werden alphanumerische Werte oder Datums- und Zeitwerte in das SELECT-Statement eingebunden, müssen diese in Hochkommata eingebettet werden.

Anstatt variable Werte direkt ins SELECT-Statement einzubinden, können sie auch – sofern es sich weder um Dateien/Tabellen/Views noch um Bibliotheksangaben handelt – in Form von Parameter-Markern (?) im SELECT-Befehl hinterlegt werden.

Der komplett aufbereitete Statement-String (mit oder ohne Parameter-Marker) muss mit Hilfe des SQL-Befehls PREPARE in ein ausführbares SQL-Statement konvertiert werden (Prepared Statement). Sofern die Konvertierung nicht erfolgreich war (z. B. aufgrund eines Syntaxfehlers) erfolgt kein Programmabbruch, stattdessen wird lediglich ein Fehlercode in der SQLCA ausgegeben. Dieser Fehlercode sollte nach Ausführung des SQL-Statements PREPARE unbedingt geprüft werden und ggf. sollten entsprechende Aktionen eingeleitet werden.

Für das **Prepared Statement** muss ein innerhalb der Quelle **eindeutiger Name** festgelegt werden. Der Name muss auch dann eindeutig sein, wenn in der Quelle mehrere interne oder exportierte Prozeduren hinterlegt sind, in denen dynamische SQL-Statements (SELECT oder sonstige SQL-Statements) verwendet werden. Da es sich bei diesem eindeutigen Namen um keine Host-Variable handelt, darf auch kein führender Doppelpunkt angegeben werden.

```
/Free
   MySelect = 'SELECT-Statement';

   //Syntax
   Exec SQL Prepare MyDynSQL from :MySelect;

   If SQLCODE < *Zeros;        //Fehler
      //Fehlerbehandlung
   EndIf;
```

SQL-Statement PREPARE bei Cursor-Verarbeitung

3.2.5.17.1.2 SQL-Befehl DECLARE CURSOR für dynamisches SQL

Über das DECLARE CURSOR-Statement wird das im PREPARE-Statement konvertierte SELECT-Statement einem eindeutigen Cursor zugeordnet.

Innerhalb einer Quelle und auch innerhalb einer Prozedur können mehrere unabhängige Cursor für statisches oder dynamisches SQL definiert werden. Für jeden Cursor muss innerhalb der Quelle ein eindeutiger Name vergeben werden. Doppelte Cursor-Namen sind auch in Quellen nicht erlaubt, in denen nur interne und exportierte Prozeduren (Schlüsselwort NOMAIN in den H-Bestimmungen) hinterlegt sind.

Anmerkung:

Das DECLARE CURSOR-Statement **muss innerhalb der Quelle vor allen anderen** den Cursor betreffenden SQL-Statements (OPEN, FETCH, CLOSE) angegeben werden.

Das DECLARE CURSOR-Statement für dynamisches SQL muss **innerhalb der Quelle nach dem PREPARE-Statement,** das das zugehörige SQL-Statement konvertiert, angegeben werden.

Da der SQL-Precompiler die Quelle ohne Rücksicht auf die Reihenfolge scannt, in der die SQL-Statements ausgeführt werden, muss die Reihenfolge, in der die SQL-Statements angegeben werden, unbedingt eingehalten werden, anderenfalls wird die Kompilierung vom SQL-Precompiler abgebrochen.

Auch bei der Verarbeitung von dynamischem SQL können serielle und Scroll-Cursor definiert werden. Wie bei der statischen Cursor-Verarbeitung wird auch bei dynamischem SQL per Default ein serieller Cursor definiert.

Die Definition eines Scroll-Cursors erfolgt wie bei statischem SQL durch die Angabe des Schlüsselworts SCROLL innerhalb des DECLARE CURSOR-Statements. Sofern ein Update über einen dynamischen Scroll-Cursor erfolgen soll, muss vor SCROLL zusätzlich DYNAMIC angegeben werden.

Um zu vermeiden, dass ein dynamischer Cursor bei Ausführung unter Commitment-Steuerung während der Ausführung der Befehle COMMIT oder ROLLBACK geschlossen wird, kann wie beim statischen Cursor das Schlüsselwort WITH HOLD innerhalb des DECLARE CURSOR-Statements angegeben werden.

```
/Free
   //SELECT-Statement - aufbereiten und konvertieren
   MySelect = 'SELECT-Statement';
   Exec SQL Prepare MyDynSQL from :MySelect;

   If SQLCODE < *Zeros;        //Fehler
      //Fehlerbehandlung
   EndIf;

   //Syntax - DECLARE CURSOR
   EXEC SQL   DECLARE  MyCursor ((DYNAMIC) SCROLL) CURSOR
                       (WITH HOLD)
                  FOR  MyDynSQL;
```

Syntax DECLARE CURSOR für dynamisches SQL

3.2.5.17.1.3 Dynamisches SQL mit fixer Auswahlliste – Beispiel

Die Schritte, die zur Definition und Verarbeitung eines Cursors mit dynamischem SQL und fixer Auswahlliste notwendig sind, werden nun in einer Prozedur hinterlegt und ausgeführt.

Das folgende Beispiel zeigt eine Prozedur, mit deren Hilfe die Jahresumsätze pro Artikel und Kunde kumuliert und die Ergebnissätze mit dem OpCode DSPLY angezeigt werden.

Über einen Parameter kann die Sortierreihenfolge, in der das Ergebnis ausgegeben werden soll, vorgegeben werden. Für das Beispiel sind drei unterschiedliche Sortierungen vorgesehen:

- Parameterwert A
  Die Ausgabe erfolgt sortiert nach Artikel-Nr., Kunden-Nr. und Jahr.
- Parameterwert K
  Die Ausgabe erfolgt sortiert nach Kunden-Nr., Jahr und Artikel-Nr.
- Parameterwert J / Default
  Die Ausgabe erfolgt nach Jahr, Kunden-Nr. und Artikel-Nr.

Das komplette SQL-SELECT-Statement – mit Ausnahme der Sortierung – wird zunächst in der Variablen CmdSQL hinterlegt. Unabhängig von der Sortierung der Datensätze werden die Spalten immer in der Reihenfolge Jahr, Kunden-Nr., Artikel-Nr. und Summe selektiert. Dadurch können die Datensätze bei Ausführung des SQL-Befehls FETCH direkt in die Datenstruktur DSCsrC1, in der die Unterfelder in der vorgegebenen Reihenfolge definiert sind, ausgegeben werden.

Abhängig vom übergebenen Parameterwert werden dem Statement-String die gewünschten ORDER BY-Anweisungen hinzugefügt.

Da ein SQL-Statement nur dann konvertiert und zugeordnet werden kann, wenn der Cursor, zu dem die Zuordnung erfolgen soll, geschlossen ist, wird zunächst versucht, den Cursor zu schließen. War der Cursor nicht geschlossen (z. B. bei einer vorherigen Ausführung wurde die Prozedur verlassen, ohne dass der Cursor geschlossen wurde), so ist er jetzt geschlossen und einer neuen Konvertierung steht nichts mehr im Weg.

Über den SQL-Befehl PREPARE wird der Statement-String in ein ausführbares SQL-Statement konvertiert. Nach Ausführung dieses Statements wird geprüft, ob der SQL-Befehl erfolgreich ausgeführt werden konnte. Im Fehlerfall erfolgt in dem Beispiel keine besondere Aktion, d. h. es wird lediglich die Prozedur verlassen.

Nachdem der String erfolgreich in ein ausführbares SQL-Statement konvertiert wurde, wird über den SQL-Befehl DECLARE CURSOR ein serieller Cursor definiert. Dabei wird der im SQL-Befehl PREPARE festgelegte Name für das Prepared Statement anstelle des SELECT-Statements angegeben.

Der deklarierte Cursor wird geöffnet und anschließend der SQLCODE darauf geprüft, ob das Open-Statement erfolgreich ausgeführt werden konnte. Im Fehlerfall wird lediglich die Prozedur verlassen.

Nach erfolgreichem Öffnen des Cursors werden die im Result-Set ausgegebenen Zeilen einzeln in einer Verarbeitungsschleife über den SQL-Befehl FETCH in die Datenstruktur DSCSRC1 übertragen. Unmittelbar nach dem FETCH-Statement wird geprüft, ob das Ende des Result-Sets erreicht wurde (SQLCODE = 100) oder ob ein Fehler aufgetreten ist (SQLCODE < 0). In beiden Fällen wird die Verarbeitungsschleife verlassen.

Konnte ein Datensatz eingelesen werden, wird der Inhalt der Datenstruktur DSCSRC1 als String aufbereitet und über den OpCode DSPLY angezeigt.

Nachdem alle Datensätze verarbeitet wurden, oder sofern ein Fehler aufgetreten ist, wird der Cursor über den SQL-Befehl CLOSE geschlossen und die Prozedur verlassen.

```
P DspUmsKundeArtNrSort...
P                 B                   Export

D DspUmsKundeArtNrSort...
D                 PI
D   ParSort                      1A   Const

D CmdSQL          S           1024A   Varying

D DSCsrC1         DS
D   CsrC1Jahr                    4P 0
D   CsrC1Kunde                  15A
D   CsrC1ArtNr                  22A
D   CsrC1Umsatz                 11P 2

D DsplyText       S             50A
 //-------------------------------------------------------------------------
 /Free
    //Statement-String aufbereiten
    CmdSQL = 'Select   Year(VerkDatum), KundeNr, ArtikelNr, Sum(Wert) +
                 From  Umsatz +
                 Group by KundeNr, ArtikelNr, Year(VerkDatum) +
                 Order By ';

    //Unterschiedliche Sortierung wird durch ParSort gesteuert
    Select;
    When ParSort = 'A';
         CmdSQL += 'ArtikelNr, Year(VerkDatum), KundeNr';
    When ParSort = 'K';
         CmdSQL += 'KundeNr, Year(VerkDatum), ArtikelNr';
    Other;
         CmdSQL += 'Year(VerkDatum), KundeNr, ArtikelNr';
    EndSL;

    //Statement-String konvertieren -> Prepared Statement
    Exec SQL Close CsrC1;
    Exec SQL  Prepare DynCsrC1 From :CmdSQL;
    If SQLCODE < *Zeros;
       //Fehlerbehandlung
       Return;
    EndIf;

    //Cursor definieren und öffnen
    Exec SQL  Declare CsrC1 Cursor For DynCsrC1;
```

3.2.5
Seite 122

```
     Exec SQL Open CsrC1;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung
        Return;
     EndIf;

     DOU 1=0;
         Exec SQL Fetch Next From CsrC1 into :DSCsrC1;
         If SQLCODE = 100;
            Leave;
         ElseIf SQLCODE < *Zeros;
            //Fehlerbehandlung
            Leave;
         EndIf;

         DsplyText = %Char(CsrC1Jahr)   + ' / ' + %Trim(CsrC1Kunde) +' / '+
                     %Char(CsrC1Umsatz) + ' / ' + %Trim(CsrC1ArtNr);
        Dsply DsplyText;
     EndDo;

     EXEC SQL  Close CsrC1;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung
        Return;
     EndIf;
     Return;

  /End-Free
P DspUmsKundeArtNrSort...
P                 E
 ********** Quelle EMBCSRDS1
```

Dynamisches SQL mit Cursor und fixer SELECT-Liste (Quelle EMBCSRDS1)

3.2.5.17.1.4 Parameter-Marker – Verwendung von Variablen in dynamischem SQL

Sofern das gleiche SQL-SELECT-Statement mehrfach – jedoch für unterschiedliche Werte, die in den WHERE-Bedingungen angegeben werden können – ausgeführt werden soll, muss das SQL-Statement nicht für jede Ausführung konvertiert werden.

Der SQL-Befehl PREPARE muss nur beim ersten Aufruf ausgeführt werden oder sofern sich das SELECT-Statement verändert (z. B. durch eine andere Sortierung oder den Zugriff auf eine andere Datei/Tabelle/View oder Bibliothek).

Die Variablen-Werte können in Form von Parameter-Markern (?) in den WHERE-Bedingungen des SELECT-Statements hinterlegt werden.

Beim Öffnen des Cursors werden die Parameter-Marker durch die in den entsprechenden Host-Variablen enthaltenen Werte ersetzt.

3.2.5.17.1.4.1 SQL-Befehl OPEN CURSOR in Verbindung mit Parameter-Markern

Sofern Parameter-Marker zum Setzen von Werten in den WHERE-Bedingungen verwendet wurden, müssen die Host-Variablen, die die variablen Werte enthalten, im OPEN-Statement in einer USING-Anweisung aufgelistet werden.

Die Anzahl der Parameter-Marker muss mit der Anzahl der angegebenen Host-Variablen übereinstimmen. Der Wert der ersten Variable wird dem ersten Parameter-Marker zugeordnet, der Wert der zweiten Variable dem zweiten Parameter-Marker usw.

3.2.5
Seite 124

```
/Free
   //Statement-String aufbereiten und konvertieren
   MySELECT = 'Select  ...  +
                  From  ...  +
                  Where      Fld1 = ?
                        And  Fld2 between ? and ?
                        Or  (FLD3 in (?, ?, ?) or Fld4 >= ?
                        ... ';
   Exec SQL Prepare MyDynSQL From :MySELECT;
   //Fehlerbehandlung

   Exec SQL  Declare CsrDyn01 For MyDynSQL;
   ...

   //Syntax OPEN für Parameter-Marker
   Exec Open CsrDyn01 Using :HostVar1, :HostVar2, ...
      :HostVarN
   //Fehlerbehandlung

   ... Cursor-Verarbeitung
```

Syntax SQL-Statement OPEN zur Zuweisung von Parameter-Markern

3.2.5.17.1.4.2 Parameter-Marker in dynamischer Cursor-Verarbeitung – Beispiel

Das folgende Beispiel zeigt die Prozedur (DspUmsKundeArtNrSort1), in der Umsatzdaten kumuliert und angezeigt werden.

Die Prozedur stellt eine Erweiterung der in Abschnitt „Dynamisches SQL mit fixer Auswahlliste – Beispiel“ vorgestellten Prozedur DspUmsKundeArtNrSort dar.

An die Prozedur DspUmsKundeArtNrSort1 können insgesamt drei Parameter übergeben werden. Im ersten Parameter wird die Sortierreihenfolge übergeben, in der die Datensätze ausgegeben werden sollen. Bei den beiden anderen Parametern handelt es sich um optionale Parameter, bei denen wahlweise das Jahr und/oder der Kunde übergeben werden können, für das/den die Umsätze ermittelt werden sollen.

Das SELECT-Statement wird aufbereitet. Für alle Varianten werden jeweils die gleichen Spalten, das Jahr, der Kunde, die Artikel-Nr. und der kumulierte Wert aus der Umsatzdatei in der gleichen Reihenfolge ausgegeben.

Abhängig von den übergebenen optionalen Parametern werden die WHERE-Bedingungen gebildet. Das Jahr und die Kunden-Nr. werden in den WHERE-Bedingungen als Parameter-Marker hinterlegt.

Basierend auf der übergebenen Sortierreihenfolge wird dem SQL-SELECT-Statement eine entsprechende ORDER-BY-Anweisung hinzugefügt.

Nach der Aufbereitung des SELECT-Statements (Statement String) wird zunächst der Cursor geschlossen, um sicherzustellen, dass das aufbereitete SELECT-Statement in ein ausführbares SQL-Statement konvertiert und/oder der Cursor problemlos geöffnet werden kann.

Das aufbereitete SQL-Statement wird anschließend mit dem bei der vorangegangenen Ausführung gesicherten SQL-Statement verglichen. Beim ersten Aufruf (es liegt noch keine gesicherte Version des SELECT-Statements vor) oder sofern das SELECT-Statement von der gesicherten Version abweicht, wird das SELECT-Statement mit Hilfe des SQL-Befehls PREPARE in ein ausführbares SQL-Statement konvertiert. Sofern das SELECT-Statement der gesicherten Version entspricht, wird das PREPARE-Statement nicht erneut ausgeführt.

Unmittelbar nach dem PREPARE-Statement wird geprüft, ob das SELECT-Statement konvertiert werden konnte. Im Fehlerfall wird das zuvor gesicherte SELECT-Statement (Variable PrvCmdSQL) initialisiert, damit beim nächsten Ablauf das PREPARE-Statement zwingend ausgeführt werden muss. Nach der erfolgreichen Ausführung des SQL-Befehls PREPARE wird der aktuelle Statement-String (Variable CmdSQL) in die statische Variable PrvCmdSQL übertragen, um den Vergleich in einer zukünftigen Ausführung zu ermöglichen.

Ergänzung 21/2012

3.2.5
Seite 126

Das konvertierte SQL-Statement DynCsrC2 wird im DECLARE CURSOR-Statement dem Cursor CsrC2 zugeordnet.

Der Cursor wird abhängig von den übergebenen Parametern bzw. der Anzahl der im SELECT-Statement hinterlegten Host-Variablen geöffnet. Sofern Kunden-Nr. und Jahr übergeben wurden, werden beide Host-Variablen im OPEN-Statement angegeben. Sofern nur einer der beiden Parameter übergeben wurde, wird jeweils nur diese Host-Variable im OPEN-Statement angegeben. Wurden weder die Kunden-Nr. noch das Jahr übergeben, wird der Cursor ohne Angabe einer Host-Variable geöffnet. Nach Ausführung des OPEN-Statements wird geprüft, ob der Befehl erfolgreich ausgeführt werden konnte. Im Fehlerfall wird die Variable PrvCmdSQL (vorheriges SQL-Statement) initialisiert und die Prozedur verlassen.

Die Datensätze werden in einer Verarbeitungsschleife einzeln über den SQL-Befehl FETCH empfangen und in die Datenstruktur DSCsrC2 übertragen. Unmittelbar nach dem FETCH-Statement wird geprüft, ob das Ende des Result-Sets erreicht wurde (SQLCODE = 100) oder ob ein Fehler aufgetreten ist (SQLCODE < *Zeros). In beiden Fällen wird die Verarbeitungsschleife verlassen. Im Fehlerfall wird zusätzlich die Variable PrvCmdSQL (vorheriges SQL-Statement) initialisiert.

Sofern ein Datensatz empfangen werden konnte, werden die ausgegebenen Werte in einem String aufbereitet und über den OpCode DSPLY angezeigt.

Nachdem alle Datensätze verarbeitet wurden oder sofern ein Fehler aufgetreten ist, wird der Cursor mit Hilfe des SQL-Befehls CLOSE geschlossen und die Prozedur anschließend beendet.

```
P DspUmsKundeArtNrSort1...
P                 B                   Export

D DspUmsKundeArtNrSort1...
D                 PI
D   ParSort                      1A   Const
D   ParJahr                      4P 0 Const Options(*NoPass)
D   ParKunde                    15A   Const Options(*NoPass)

D DSCsrC2         DS
D   CsrC2Jahr                    4P 0
D   CsrC2Kunde                  15A
D   CsrC2ArtNr                  22A
D   CsrC2Umsatz                 11P 2

D DsplyText       S              50A
D isWhere         S                N

D CmdSQL          S            1024A   Varying
D PrvCmdSQL       S            1024A   Varying   Static
//-------------------------------------------------------------------------
 /Free

    //Aufbereiten SQL Statement
    CmdSQL = 'Select   Year(VerkDatum), KundeNr, ArtikelNr, Sum(Wert) +
                 From  Umsatz';

    //Variable WHERE-Bedingungen abh. von übergebenen Parametern
    //-> Einbindung von Parameter-Markern
    If %Parms >= %ParmNum(ParKunde) and %Len(%Trim(ParKunde)) <> *Zeros;
       CmdSQL += ' Where KundeNr = ? ';
       isWhere = *On;
    EndIf;

    If %Parms >= %ParmNum(ParJahr) and ParJahr > 2000;
       If isWhere;
          CmdSQL += ' And ';
       Else;
          CmdSQL += ' Where ';
       EndIf;
       CmdSQL += ' Year(VerkDatum) = ? ';
    EndIf;

    CmdSQL += '  Group by KundeNr, ArtikelNr, Year(VerkDatum) +
                 Order By ';
```

3.2.5
Seite 128

```
Select;
When ParSort = 'A';
     CmdSQL += 'ArtikelNr, Year(VerkDatum), KundeNr';
When ParSort = 'K';
     CmdSQL += 'KundeNr, Year(VerkDatum), ArtikelNr';
Other;
     CmdSQL += 'Year(VerkDatum), KundeNr, ArtikelNr';
EndSL;
CmdSQL = %Trim(CmdSQL);

Exec SQL Close CsrC2;

//PREPARE nur bei anderem SELECT-Statement gegenüber vorh. Aufruf
If CmdSQL <> PrvCmdSQL;                          //Abw. SELECT-Statement
   Exec SQL  Prepare DynCsrC2 From :CmdSQL;
   If SQLCODE < *Zeros;
      //Fehlerbehandlung
      Return;
   EndIf;
   PrvCmdSQL = CmdSQL;                           //Statement sichern
EndIf;

Exec SQL  Declare CsrC2 Cursor For DynCsrC2;

//Öffnen des Cursors mit Hostvariablen, abh. von übergebenen Parametern
If     %Parms >= %ParmNum(ParKunde) and %Len(%Trim(ParKunde)) > *Zeros
   and ParJahr > 2000;
   Exec SQL Open CsrC2 using :ParKunde, :ParJahr;
ElseIf %Parms >= %ParmNum(ParKunde) and %Len(%Trim(ParKunde)) > *Zeros
   and ParJahr <= 2000;
   Exec SQL Open CsrC2 using :ParKunde;
ElseIf %Parms >= %ParmNum(ParJahr) and ParJahr >= 2000;
   Exec SQL Open CsrC2 using :ParJahr;
Else;
   Exec SQL Open CsrC2;
EndIf;

If SQLCODE < *Zeros;
   //Fehlerbehandlung
   Clear PrvCmdSQL;
   Return;
EndIf;
```

```
     //Verarbeiten des Result-Sets
     DOU 1=0;
         Exec SQL Fetch Next From CsrC2 into :DSCsrC2;
         If SQLCODE = 100;
            Leave;
         ElseIf SQLCODE < *Zeros;
            //Fehlerbehandlung
            Clear PrvCmdSQL;

            Leave;
         EndIf;

         DsplyText = %Char(CsrC2Jahr)   + ' / ' + %Trim(CsrC2Kunde) +' / '+
                     %Char(CsrC2Umsatz) + ' / ' + %Trim(CsrC2ArtNr);
        Dsply DsplyText;
     EndDo;

     EXEC SQL  Close CsrC2;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung
        Clear PrvCmdSQL;
        Return;
     EndIf;
     Return;

  /End-Free
 P DspUmsKundeArtNrSort1...
 P                 E
  ************** Quelle EMBCSRDS1
```

Dynamisches SQL mit Cursor und Parameter-Markern

3.2.5.17.1.5 Dynamisches SQL mit fixer Auswahlliste oder statisches SQL?

In den meisten Fällen, in denen dynamisches SQL zum Einsatz kommt, wird eine statische SELECT-Liste verwendet. Die unterschiedlichen SQL-Statements unterscheiden sich lediglich durch unterschiedliche WHERE-Anweisungen und abweichende Sortierungen voneinander.

Die Frage, die sich dabei stellt, lautet: Ist dynamisches SQL überhaupt notwendig?

Statisches SQL hat gegenüber dynamischem SQL einige Vorteile:

- Da das SQL-Statement bereits zur Kompilierungszeit bekannt ist, kann die Syntaxprüfung bereits bei der Kompilierung erfolgen.
  Probleme, die zur Laufzeit durch unsaubere Codierung auftreten können (z. B. beim Aufbereiten des Strings wurde ein Leerzeichen, Hochkomma oder Komma vergessen), können dadurch vermieden werden.
- Mit Hilfe von Host-Variablen können auch SELECT-Statements bzw. die Ergebnisse aus diesen SQL-Befehlen sehr variabel gestaltet werden. Über CASE-Anweisungen besteht sowohl in WHERE-Anweisungen als auch in ORDER BY-Anweisungen die Möglichkeit, auf unterschiedliche Situationen unterschiedlich zu reagieren.

Bei statischem SQL wird der Zugriffsplan/Access Plan, der zur Ausführung des statischen SQL-Statements verwendet wird, im (Service-)Programm-Objekt gespeichert.

Bei der wiederholten Ausführung der statischen SQL-Statements kann der gespeicherte Zugriffsplan/Access Plan validiert werden und muss nicht neu generiert werden. Sofern der Zugriffsplan/Access Plan aufgrund von geänderten Datenkonstellationen aktualisiert wird, wird der neue Zugriffsplan ebenfalls im (Service-)Programm-Objekt hinterlegt.

Das folgende Beispiel zeigt, dass in den meisten Fällen dynamisches SQL vermieden und stattdessen statisches SQL eingesetzt werden kann.

In dem Beispiel werden wiederum die Umsätze pro Jahr und Kunde kumuliert und in unterschiedlichen Sortierungen ausgegeben. Im Gegensatz zu dem Beispiel in Abschnitt „Parameter-Marker in dynamischer Cursor-Verarbeitung“ wird kein dynamisches, sondern statisches SQL verwendet. Ebenso wird das komplette Result-Set mit einem einzigen FETCH-Statement in den Ausgabe-Parameter übertragen, in die Array-Datenstruktur POutUmsatz.

Das Jahr, die Kunden-Nr. und die Sortierung können als optionale Parameter an die Prozedur übergeben werden.

Die notwendige Dynamik wird mit Hilfe von CASE-Anweisungen erreicht, die in den WHERE- und ORDER-BY-Anweisungen hinterlegt wurden:

- Sofern ein Jahr größer als das Jahr 2000 übergeben wurde, werden nur die Werte für dieses Jahr ausgewählt. Sofern ein kleineres Jahr oder der Parameter nicht angegeben wurde, werden alle Zeilen aus allen Jahren ausgegeben. Das wird dadurch erreicht, dass der Spaltenwert mit sich selber verglichen wird (**JAHR** = Case When … Else **JAHR** End)
- Sofern eine Kunden-Nr. übergeben wurde, werden nur die Umsätze dieses Kunden ausgewählt. Wurde keine Kunden-Nr. angegeben, werden die Umsätze aller Kunden ausgegeben.
- Wurde die Sortierung 1 angegeben, werden die Datensätze nach Jahr und Kunden-Nr. sortiert.
- Wurde die Sortierung 2 angegeben, werden die Datensätze nach Jahr, Umsatzwert absteigend und Kunden-Nr. sortiert.
- Wurde die Sortierung 3 angegeben, werden die Datensätze nach Kunden-Nr. und Jahr sortiert.
- Für jede Sortierungsvariante und jede Spalte, nach der sortiert werden kann, wird eine CASE-Anweisung angegeben. Sofern die Bedingung nicht zutrifft, wird NULL ausgegeben. NULL-Werte in den ORDER BY-Anweisungen werden bei der Sortierung ignoriert.

3.2.5
Seite 132

```
D RefDSUmsatz     DS                  Qualified
D    Jahr                          4P 0
D    KundeNr                      15A   Varying
D    Name                         35A   Varying
D    Wert                         11P 2
 ******************************************************************************
P UmsatzJahrKunde...
P                 B

D UmsatzJahrKunde...
D                 PI             5I 0
D  POutUmsatz                           LikeDS(RefDSUmsatz) Dim(100)
D  PInJahr                       4P 0 Const Options(*NoPass)
D  PInKundeNr                   15A   Const Options(*NoPass)
D  PInSort                       5I 0 Const Options(*NoPass)

D ParJahr         S              4P 0
D ParKundeNr      S             15A
D ParSort         S              5I 0 inz(1)

D RtnNbrRows      S              5I 0
 *-----------------------------------------------------------------------------
 /Free
   Clear POutUmsatz;
   If %Parms >= %ParmNum(PInJahr);
      ParJahr = PInJahr;
   EndIf;

   If %Parms >= %ParmNum(PInKundeNr);
      ParKundeNr = PInKundeNr;
   EndIf;

   If %Parms >= %ParmNum(PInSort) and PInSort > *Zeros and PInSort <= 3;
      ParSort = PInSort;
   EndIf;

   Exec SQL  Declare CsrC01 Cursor For
                 Select Jahr, KundeNr, Name, Wert
                    From UmsKndNamJ
                    Where     Jahr = Case When :ParJahr > 2000
                                          Then :ParJahr
                                          Else Jahr End
                          and KundeNr = Case When Trim(:ParKundeNr) <> ''
                                             Then :ParKundeNr
                                             Else KundeNr End
```

```
                    Order By Case When :ParSort in (1, 2)
                                  Then Jahr Else NULL End,
                             Case When :ParSort = 1
                                  Then KundeNr Else NULL End,
                             Case When :ParSort = 2
                                  Then Wert Else NULL End Desc,
                             Case When :ParSort = 2
                                  Then KundeNr Else NULL End,
                             Case When :ParSort = 3
                                  Then KundeNr else NULL End,
                             Case When :ParSort = 3
                                  Then Jahr else NULL End
                  For Fetch Only;

   Exec SQL  Close CsrC01;
   Exec SQL  Open  CsrC01;
   If SQLCODE < *Zeros;
      Return SQLCODE;
   EndIf;

   Exec SQL  Fetch Next From CsrC01 For 100 Rows into :POutUmsatz;
   If SQLCODE < *Zeros;
      RtnNbrRows = SQLCODE;
   ElseIf SQLCODE <> 100;
      RtnNbrRows = SQLER3;
   EndIf;

   Exec SQL  Close CsrC01;
   If SQLCODE < *Zeros;
      Return SQLCODE;
   EndIf;

   Return RtnNbrRows;
 /End-Free
P UmsatzJahrKunde...
P                 E
 ************ Quelle EMBCSRCASE
```

Statisches SQL mit CASE-Anweisungen

3.2.5.17.2 Dynamisches SQL mit variabler Auswahlliste

Bei diesen speziellen dynamischen SELECT-Statements kann jede beliebige Spalte/Feld aus jeder beliebigen Tabelle/Datei/View ausgewählt werden. Weiterhin ist es möglich, beliebige neue Spalten/Felder zu generieren (z. B. wird in einer zusätzlichen Spalte der Gesamtwert über Menge * Preis ermittelt).

Da zur Kompilierungszeit weder Informationen über die ausgewählten Dateien/Tabellen/Views, die ausgewählten Spalten oder die zusätzlich generierten Spalten vorliegen, können weder Host-Strukturen noch Host-Variablen angelegt werden, in die die Daten ausgegeben werden könnten.

Um die im Result-Set ausgegebenen Daten empfangen und verarbeiten zu können, muss mit einer SQL Descriptor Area (SQLDA) gearbeitet werden. In der SQL Descriptor Area werden Pointer auf die ausgegebenen Daten verwaltet. Ebenso werden Informationen wie der Datentyp, die Länge und die Anzahl Nachkommastellen ausgegeben.

Die verwendete SQL Descriptor Area muss dem dynamischen SQL-Statement zugewiesen werden. Die Zuweisung kann auf zwei Arten erfolgen:

SQL-Befehl DESCRIBE
Die Zuordnung der SQL Descriptor Area zu dem konvertierten SQL-SELECT-Statement (Prepared Statement) erfolgt über den SQL-Befehl DESCRIBE.

SQL-Befehl PREPARE
Anstatt einen separaten Befehl auszuführen, kann die SQL Descriptor Area auch direkt im PREPARE-Statement zugewiesen werden.

Da die Daten des Result-Sets weder in eine vordefinierte Host-Struktur noch in vordefinierte Host-Variablen ausgegeben werden können, muss die Ausgabe der Daten in die SQL Descriptor Area erfolgen. Aus diesem Grund, muss die zuvor dem dynamischen SELECT-Statement zugewiesene SQL Descriptor Area im FETCH-Statement angegeben werden.

3.2.5.17.2.1 SQL Descriptor Area (SQLDA)

Bei dynamischem SQL mit Cursor-Verarbeitung und variabler Auswahlliste ist die Verwendung einer SQL Descriptor Area erforderlich.

In der SQL Descriptor Area ist die Struktur der zu empfangenden Daten hinterlegt, d. h. Anzahl, Datentyp und Länge. Weiterhin werden Pointer auf die empfangenen Daten ausgegeben.

Im Gegensatz zur SQL Communications Area, die zumindest in RPG automatisch vom SQL-Precompiler eingebunden wird, muss die SQL Descriptor Area explizit über eine SQL INCLUDE-Anweisung eingebunden werden.

Zusätzlich muss die **Konstante SQL_NUM** definiert werden, in der die maximale Anzahl an Spalten hinterlegt wird, die über das dynamische SELECT-Statement ausgewählt werden können. Die Konstante SQL_NUM wird in der durch die Copy-Strecke SQLDA eingebundenen Datenstruktur verwendet, um für jedes mögliche Vorkommen einen Speicherbereich zu reservieren.

Die Konstante SQL_NUM und die SQL Descriptor Area werden am Ende der globalen D-Bestimmungen wie folgt angegeben:

```
D ... Globale D-Bestimmungen

D SQL_NUM         C                   Const(x)

C/Exec SQL   Include SQLDA
C/End-Exec

... Globale C-Bestimmungen oder P-Bestimmung
```

Einbinden der SQL Descriptor Area über den SQL-Befehl INCLUDE

Die über den SQL-Befehl INCLUDE eingebundene Copy-Strecke hat folgenden Aufbau:

3.2.5
Seite 136

```
 * Konstante SQL_NUM -> muss manuell definiert werden!
D SQL_NUM         C                   Const(x)

D*      SQL DESCRIPTOR AREA
D SQLDA           DS
D  SQLDAID                       8A
D  SQLDABC                      10I 0
D  SQLN                          5I 0
D  SQLD                          5I 0
D  SQL_VAR                      80A   DIM(SQL_NUM)
D                        33     48*
D                        49     64*
D*
D SQLVAR          DS
D  SQLTYPE                       5I 0
D  SQLLEN                        5I 0
D  SQLRES                       12A
D  SQLINFO1                     16A
D   SQLDATA                       *   OVERLAY(SQLINFO1:1)
D   SQL_RESULT_SET_LOCATOR...
D                               20I 0 OVERLAY(SQLINFO1:1)
D  SQLINFO2                     16A
D   SQLIND                        *   OVERLAY(SQLINFO2:1)
D   SQL_ROW_CHANGE...
D                               10I 0 OVERLAY(SQLINFO2:1)
D   SQL_RESULT_SET_ROWS...
D                               10I 0 OVERLAY(SQLINFO2:1)
D  SQLNAMELEN                    5I 0
D  SQLNAME                      30A
D*  END OF SQLDA
```

SQLDA – SQL Descriptor Area

Datenstruktur SQLDA

SQL Descriptor Area – Der Header-Bereich besteht aus vier Datenstruktur-Unterfeldern, in denen die folgenden Informationen hinterlegt sind:

- SQLDAID enthält fix 'SQLDA'.
- SQLDABC enthält die Länge der SQLDA-Datenstruktur.
- SQLN wird vom Datenbanken-Manager nicht verändert.
- SQLD gibt die Anzahl der tatsächlich ausgewählten Spalten aus.
- SQL_VAR Für jede mögliche Spalte wird (abhängig vom Inhalt der Konstante SQL_NUM) ein 80-stelliger Speicherbereich reserviert.

SQLVAR – Sofern es sich bei der ausgewählten Spalte weder um ein Large Object noch einen User Defined Data Type handelt, hat der 80-stellige reservierte Speicherbereich (SQL_VAR) den Aufbau der Datenstruktur SQLVAR.

Die Datenstruktur SQLVAR hat die folgenden Unterfelder:

- SQLTYPE enthält den Datentypen der Spalte. Der Datentyp wird als 2-Byte Binärwert (Small Integer/5 I 0) ausgegeben. Für jeden Datentyp werden zwei Binärwerte zur Verfügung gestellt, für NULL-fähige und für nicht NULL-fähige Ausgaben.
- SQLLEN enthält die Länge der ausgegebenen Daten.

3.2.5
Seite 138

Die folgende Tabelle gibt einen Überblick über die Datentypen und die Längen, die in der SQLDA verwendet werden.

SQL – Descriptor Area – Datentypen / Längen-Definitionen			
Datentypen		Längen	
SQLTYPE	Beschreibung	SQLLEN	Beschreibung
384 / 385	Datum		10 Byte
388 / 389	Zeit		8 Byte
392 / 393	Zeitmarke		26 Byte
448 / 449	alphanumerisch mit variabler Länge (SBCS)	> 254	255 – 32766 Zeichen
456 / 457		<= 254	1 – 254 Zeichen
452 / 453	alphanumerisch mit fixer Länge (SBCS)	<= 32766	1 – 32766 Zeichen
464 / 465	graphisch mit variabler Länge (DBCS)	<= 127	1 – 127 Zeichen
472 / 473		<= 16383	128 – 16383 Zeichen
468 / 469	graphisch mit fixer Länge (DBCS)	<= 16383	1 – 16383 Zeichen
480 / 481	Fließkomma	4	4 Byte für einfache Genauigkeit
		8	8 Byte für doppelte Genauigkeit
484 / 485	Gepackt numerisch		1.Byte = Länge / 2.Byte = Dezimal-Positionen
488 / 489	Gezont numerisch		1.Byte = Länge / 2.Byte = Dezimal-Positionen
492 / 493	Big Integer / Ganzzahl		8 Byte
496 / 497	Integer / Ganzzahl		4 Byte
500 / 501	Small Integer / Ganzzahl		2 Byte

SQL-Descriptor Area – Datentypen und Längen

- SQLRES 12 Byte reservierter Bereich, der ansonsten nicht verwendet wird.
- SQLDATA Adresse (Pointer) der im FETCH ausgegebenen Daten.
- SQL_RESULT_SET_LOCATOR wurde mit Release 7.1 zur Verarbeitung von Result-Sets, die aus Stored Procedures ausgegeben werden, eingeführt.
- SQL_RESULT_SET_LOCATOR überlagert das Unterfeld SQLDATA.
- SQLIND Pointer auf eine 2-Byte Binär-Indikator-Variable (Small Integer / 5I 0). Sofern für die Spalte ein NULL-Wert ausgegeben wurde, wird der Inhalt auf -1 gesetzt. Ansonsten wird 0 ausgegeben.

- SQL_ROW_CHANGE überlagert SQLIND und enthält Informationen über die automatische Aktualisierung von Zeitmarkenfeldern beim Schreiben oder Ändern von Datensätzen.
- SQL_RESULT_SET_ROWS überlagert SQLIND und enthält bei der dynamischen Verarbeitung von Result-Sets aus Stored Procedures die Anzahl der ausgegebenen Datensätze.
- SQLNAME enthält den (System-)Spalten-Namen, das Label (bei neu erstellten Spalten) oder die Spalten-Nr.

SQLVAR2 – Erweiterte SQLVAR-Datenstruktur

Werden Large Object-Spalten (CLOB, DBCLOB, BLOB) oder individuell definierte Datentypen (User Defined Data Types = UDT) verwendet, muss anstatt der SQLVAR-Datenstruktur die SQLVAR2-Datenstruktur verwendet werden, um auf die ausgegebenen Daten sowie die Informationen über die Spalten zugreifen zu können.

Obwohl in RPGIV bis zu 4096 Zeichen lange Variablen definiert und verwendet werden können, wurden die in anderen Sprachen (z. B. C oder Cobol) verwendeten langen Standardnamen abgekürzt. Im Gegensatz zur SQLCA, in die die langen Namen, die auch in anderen Programmiersprachen verwendet werden, integriert wurden, wurde die SQLDA nach der Umstellung von RPGIII auf RPGIV nicht angepasst.

In der erweiterten SQLVAR2-Datenstruktur sind die folgenden Unterfelder definiert:

- SQLLONGLEN (RPG: SQLLONGL) enthält die Länge der ausgegebenen Large Objects (LOB).
- SQLDATALEN (RPG: SQLDATAL) enthält die Länge (Anzahl Zeichen, nicht Anzahl Bytes) für Large Objects.
- SQLDATATYPE_NAME ist ein Feld mit variabler Länge, bestehend aus einem 4-Byte-Präfix (SQLDATATYPE_NAMEL / RPG: SQLTNAMELN und einem maximal 30-stelligen Datenfeld (SQLDATATYPE_NAMEC / RPG: SQLTNAME).
- SQLDATATYPE_NAME ist in der RPG-Datenstruktur nicht definiert.
- SQLDATATYPE_NAMEL (RPG: SQLTNAMELN) enthält die Länge des Namens, des LOBs oder des User Defined Data Types (UDT).
- Wird ein NULL-Wert ausgegeben, enthält dieses Feld die Länge 0.
- SQLDATATYPE_NAMEC (RPG: SQLTNAME) enthält entweder den voll qualifizierten Namen der UDT (User Defined Data Type), die ersten 20 Stellen eines Labels oder den Spalten-Namen.
  Längere Namen werden abgeschnitten.

Anmerkung:

Mit Release 7.1 wurde die SQL Descriptor Area erweitert, um die dynamische Verarbeitung von Result-Sets, die aus Stored Procedures zurückgegeben werden, zu ermöglichen.

Nach der Umstellung auf Release 7.1. sollten alle Module, Programme und Service-Programme, in denen die SQL Descriptor Area verwendet wird, erneut umgewandelt werden.

3.2.5.17.2.2 SQL-Befehl DESCRIBE – Zuweisung der SQL Descriptor Area

Über den SQL-Befehl DESCRIBE kann eine SQL Descriptor Area einem dynamischen SQL-Statement zugeordnet werden.

Das DESCRIBE-Statement muss innerhalb der Quelle nach dem PREPARE-Statement eingebunden werden. Der Name des SQL-Statements muss in dem vorgelagerten PREPARE-Statement festgelegt worden sein.

Wahlweise können zur Ausgabe der Feldnamen die SQL-Namen (Default) oder die Systemnamen verwendet werden.

Anmerkung:

Bei der Verwendung von System- oder SQL-Namen ist Folgendes zu beachten:
SQL-Namen können bis zu einer Länge von 128 Zeichen definiert werden, in der Descriptor Area werden die Namen jedoch auf 30 Zeichen beschnitten.
Systemnamen sind zwar auf zehn Zeichen begrenzt, es gibt jedoch keine Garantie, dass beim erneuten Erstellen eines Datenbankenobjekts wieder die gleichen Systemnamen verwendet werden.

```
/Free
   //Syntax
   Exec SQL Describe DynSQLStmt into :SQLDA
            Using Names/System Names
```

SQL-Statement DESCRIBE – Zuordnen der SQL Descriptor Area – Syntax

3.2.5
Seite 141

Im folgenden Beispiel wird die SQL Descriptor Area dem dynamischen SQL-Statement DynC1 zugeordnet, das zuvor im PREPARE-Statement in ein ausführbares SQL-Statement konvertiert wurde. Bei der Ausgabe der Daten in die SQLDA (SQL Descriptor Area) werden die System-Spalten-Namen verwendet.

```
 D SQL_NUM         C                   Const(10)

 C/Exec SQL  Include SQLDA
 C/End-Exec
  *-------------------------------------------------------
  /Free
   //Aufbereiten Statement-String in Variable CmdSQL
   ...
   //Konvertieren Statement-String in ausführbares SQL-State-
ment
    Exec SQL  Prepare DynC1 From :CmdSQL;
    If SQLCODE < *Zeros;
       //Fehlerbehandlung
    EndIf;

    //Definition und Zuordnung SQLDA zu dynamischem SQL
    Exec SQL  Describe DynC1 into :SQLDA using System Names;
    If SQLCOD < *Zeros;
       //Fehlerbehandlung
    EndIf;
```

SQL-Statement DESCRIBE – Zuordnen SQL Descriptor Area – Beispiel

3.2.5.17.2.3 SQL-Befehl PREPARE in Verbindung mit SQL Descriptor Area

Anstatt die SQL Descriptor Area dem dynamischen SQL-Statement über einen separaten SQL-Befehl zuzuweisen, kann die Zuordnung direkt im PREPARE-Statement erfolgen.

Wie auch beim DESCRIBE-Statement kann angegeben werden, ob mit SQL oder mit Systemnamen gearbeitet wird.

```
 /Free
    //Syntax
    Exec SQL Prepare DynSQLStmt into :SQLDA Using Names/
System Names
                From StatementString;
```

SQL-Statement PREPARE – Zuordnen SQL Descriptor Area – Syntax

Im folgenden Beispiel wird die SQL Descriptor Area dem dynamischen SQL-Statement DynC1 direkt im PREPARE-Statement zugeordnet. Bei der Ausgabe der Daten in die SQLDA (SQL Descriptor Area) werden die System-Spalten-Namen verwendet.

```
D SQL_NUM         C                   Const(10)
 C/Exec SQL  Include SQLDA
 C/End-Exec
  *-------------------------------------------------------
  /Free
    //Aufbereiten Statement-String in Variable CMDSQL
    ...
    //Konvertieren Statement-String in ausführbares SQL-
Statement
    Exec SQL  Prepare DynC1 into :SQLDA using System Names
from :CmdSQL;
    If SQLCODE < *Zeros;
       //Fehlerbehandlung
    EndIf;
```

SQL-Statement PREPARE – Zuordnen SQL Descriptor Area – Beispiel

3.2.5.17.2.4 SQL-Befehl FETCH in Verbindung mit SQL Descriptor Area

Sofern bei dynamischem SQL weder die zur Laufzeit ausgewählten Spalten noch deren Datentypen und Längen bekannt sind, muss mit einer SQL Descriptor Area gearbeitet werden. Die SQL Descriptor Area wird dem dynamischen SQL-Statement entweder direkt im PREPARE-Statement oder in einem separat ausgeführten DESCRIBE-Statement zugeordnet.

Die Descriptor Area muss im FETCH-Statement anstelle von Host-Variablen oder Host-Strukturen angegeben werden, um die Daten des Result-Sets zu empfangen.

Da auch beim dynamischen SQL-Statement serielle und Scroll-Cursor definiert werden können, kann auch – abhängig von der Art des Cursors – durch Angabe des entsprechenden Schlüsselworts innerhalb des Result-Sets vorwärts und rückwärts gelesen oder innerhalb des Result-Sets positioniert werden.

```
/Free
    //Syntax
    Exec SQL  FETCH Next (Prior/First/Last/Before/After/
Current/Relative n)
                  FROM MyCursor
                  INTO Descriptor :SQLDA;
```

Syntax SQL-Statement FETCH mit SQL Descriptor Area

Im folgenden Beispiel werden die einzelnen Schritte aufgezeigt, die zur Verarbeitung eines SELECT-Statements mit dynamischer Auswahlliste notwendig sind.

Die SQL Descriptor Area (SQLDA) wird über den SQL-Befehl INCLUDE in die Quelle eingebunden.

Die Zuweisung der SQL Descriptor Area erfolgt direkt bei der Konvertierung in ein ausführbares SQL-Statement im PREPARE-Befehl.

Bei der Verarbeitung werden die Daten durch die Angabe der SQL Descriptor Area im FETCH-Statement direkt in die Datenstruktur ausgegeben.

Die Verarbeitung der eingelesenen Daten kann anschließend mit Hilfe von Pointer-Handling erfolgen.

3.2.5
Seite 144

```
 D CmdSQL          S           1024A   Varying

 D SQL_NUM         C                   Const(10)
 C/Exec SQL  Include SQLDA
 C/End-Exec
  *---------------------------------------------------------
/Free
    //Aufbereiten Statement-String
    CmdSQL = 'Select ... ';
    ...
    //Konvertieren Statement-String in ausführbares SQL-
Statement
    Exec SQL  Prepare DynC1 into :SQLDA using System Names
from :CmdSQL;
    ...
    //Cursor für dynamisches Statement definieren/deklarieren
    Exec SQL  Declare C1 Cursor For DynC1;

    //Cursor öffnen
    Exec SQL Open C1;
    ...

    //Verarbeitung des Result-Sets
    DoU 1 = 0;
        EXEC SQL Fetch Next From C1 Into Descriptor :SQLDA;
        If SQLCODE = 100;
           Leave;
        ElseIf SQLCODE < *Zeros;
           //Fehlerbehandlung
        EndIf;

        //Verarbeitung der eingelesenen Daten
        ...
    EndDo;

    //Cursor schließen
    Exec SQL Close C1;
    ...
 /End-Free
 ******* Quelle EMBDYNCDA1
```

Verarbeitungsschritte bei dynamischem SQL mit variabler Auswahlliste

3.2.5.17.2.5 Dynamisches SQL mit flexibler Auswahlliste – Beispiel

Nachdem die für die Verarbeitung von dynamischem SQL mit flexibler Auswahlliste notwendigen Schritte detailliert beschrieben wurden, werden diese im folgenden Beispiel nacheinander ausgeführt.

Das folgende Beispiel zeigt die Prozedur „DspUmsKundeArtNrSortx“. An diese Prozedur wird ein Parameter „ParSort“ übergeben, über den unterschiedliche Aufbereitungen mit unterschiedlichen Auswahlfeldern, Summierungen und Sortierungen gesteuert werden können.

Insgesamt sind drei Auswahlmöglichkeiten vorgesehen:

A Auswahl der Spalten „Artikel-Nr.“, „Jahr“ und „Gesamt“. In der letzten Spalte werden die Umsätze pro Artikel-Nr. und Jahr kumuliert ausgegeben.

  Das Ergebnis wird nach Artikel-Nr. und Jahr sortiert.

J Auswahl der Spalten „Jahr“ und „Gesamt“, in der die kumulierten Jahresumsätze ausgegeben werden.

  Das Ergebnis wird nach Jahr sortiert ausgegeben.

K Auswahl der Spalten „Kunden-Nr.“, „Artikel-Nr.“, „Jahr“ und „Gesamt“. In der letzten Spalte werden die kumulierten Umsätze ausgegeben.

  Die Sortierung erfolgt nach Kunden-Nr., Artikel-Nr. und Jahr. Abhängig vom übergebenen Parameterwert wird das dynamische SELECT-Statement (Statement-String) aufbereitet.

Nach der Aufbereitung wird zunächst der Cursor geschlossen, um sicherzugehen, dass sowohl das PREPARE- als auch das OPEN-Statement erfolgreich ausgeführt werden können.

Anschließend wird der Statement-String CMDSQL mit Hilfe des PREPARE-Statements in ein ausführbares SQL-Statement konvertiert (Prepared String). Im PREPARE-Statement wird die SQL Descriptor Area SQLDA zugewiesen. Durch die Angabe von SYSTEM NAMES wird festgelegt, dass die kurzen System-Spalten-Namen verwendet werden.

Die SQLDA wurde am Ende der globalen D-Bestimmungen über den SQL-Befehl INCLUDE eingebunden. Die Konstante SQL_NUM wird in der SQLDA verwendet und muss vom Programmierer in den globalen D-Bestimmungen mit der maximal zu erwartenden Spaltenanzahl definiert werden. Für das Beispiel wird festgelegt, dass im dynamischen SQL SELECT-Statement maximal zehn Spalten ausgewählt werden können.

3.2.5
Seite 146

Nach Ausführung des PREPARE-Statements wird geprüft, ob die Konvertierung erfolgreich war. Im Fehlerfall wird lediglich die Prozedur verlassen.

Da in der Prozedur in allen dynamischen SQL-Statements nur auf die Tabelle/Datei „Umsatz“ zugegriffen wird, können auch nur die Spalten aus dieser Tabelle/Datei verwendet werden. Damit ist die Definition – d. h. Datentyp und Länge der potenziellen Ausgabefelder – bekannt. Eine komplizierte Pointer-Verarbeitung ist somit nicht erforderlich, stattdessen kann eine Datenstruktur verwendet werden, in der alle in der Tabelle/Datei vorhandenen Spalten definiert sind.

Da die SQL Descriptor Area bereits dem SQL-Statement zugewiesen wurde, werden auch alle Informationen – mit Ausnahme der Daten – in die SQL Descriptor Area übertragen. Der Speicher, in den die Daten übertragen werden, ist bereits reserviert und entsprechende Adressen sind bereitgestellt. Diese Pointer können den Feldern in der Datenstruktur zugewiesen werden.

Über eine FOR-Schleife werden die in der SQLDA definierten Spalteninformationen verarbeitet. Abhängig vom Spalten-Namen (SQLNAME) werden die Adressen der Daten (SQLDATA) den entsprechenden Unterfeldern in der Datenstruktur DSC1 zugewiesen (z. B. SQLData = %Addr(C1Artikel)).

Anschließend wird das konvertierte dynamische SQL-Statement DYNC1 im DECLARE CURSOR-Statement dem seriellen Cursor C1 zugeordnet.

Der Cursor C1 wird geöffnet. Nach dem OPEN-Statement wird geprüft, ob das Statement ordnungsgemäß verarbeitet wurde. Im Fehlerfall wird die Prozedur verlassen.

Über eine Verarbeitungsschleife werden die Datensätze des Result-Sets einzeln verarbeitet. Im FETCH-Statement werden die Daten in die SQL Descriptor Area übertragen. Da zuvor bereits eine Zuordnung der Daten-Pointer zu den Unterfeldern in der Datenstruktur DSC1 erfolgt ist, werden die Daten direkt in diese Datenstruktur-Unterfelder übertragen und können verarbeitet werden.

Nach dem FETCH-Statement wird geprüft, ob der letzte Datensatz erreicht wurde oder ob das FETCH-Statement fehlerhaft ausgeführt wurde. In beiden Fällen wird die Verarbeitungsschleife verlassen.

Wurde ein Datensatz eingelesen, werden die übertragenen Daten konvertiert und als Text aufbereitet, der mit dem OpCode DSPLY angezeigt wird.

Nachdem die Verarbeitungsschleife verlassen wurde – entweder nach Ausgabe aller Datensätze oder im Fehlerfall – wird der Cursor durch den SQL-Befehl CLOSE geschlossen und die Prozedur anschließend beendet.

```
 * Globale D-Bestimmungen
 *----------------------*
 * SQL_NUM: Konstante Verwendung in SQLDA muss manuell definiert werden.
D SQL_NUM         C                   Const(10)

C/Exec SQL  Include SQLDA
C/End-Exec
 ******************************************************************************
P DspUmsKundeArtNrSortx...
P                 B

D DspUmsKundeArtNrSortx...
D                 PI
D   ParSort                      1A   Const

D CmdSQL          S           1024A   Varying
D Index           S              3U 0

D DSC1            DS                  Inz
D   C1Kunde                     15A
D   C1Artikel                   22A
D   C1Jahr                      10I 0
D   C1Gesamt                    11P 2

D DsplyText       S              50A
 *-----------------------------------------------------------------------------
 /Free
   DsplyText = 'Sortierung : ' + ParSort;
   Dsply DsplyText;

   //Aufbereiten SQL-SELECT-Statements abh. von ParSort
   //ParSort A: Gruppierung Artikel-Nr. und Jahr
   //ParSort J: Gruppierung Jahr
   //ParSort K: Gruppierung Kunde, Artikel-Nr. und Jahr
   Select;
   When ParSort = 'A';
        CmdSQL = 'Select ArtikelNr, Year(VerkDatum) as Jahr,          +
                         Cast(Sum(Wert) as Dec(11, 2)) as Gesamt       +
                   From Umsatz                                         +
                   Group By ArtikelNr, Year(VerkDatum)                 +
                   Order By ArtikelNr, Year(VerkDatum)';

   When ParSort = 'J' or ParSort = *Blanks;
        CmdSQL = 'Select Year(VerkDatum) as Jahr,                     +
                         Cast(Sum(Wert) as Dec(11, 2)) as Gesamt       +
                   From Umsatz                                         +
```

3.2.5
Seite 148

```
                    Group By Year(VerkDatum)                              +
                    Order By Year(VerkDatum)';

     When ParSort = 'K';
          CmdSQL = 'Select KundeNr, ArtikelNr, Year(VerkDatum) as Jahr,  +
                           Cast(Sum(Wert) as Dec(11, 2)) as Gesamt       +
                    From Umsatz                                           +
                    Group By KundeNr, ArtikelNr, Year(VerkDatum)          +
                    Order By KundeNr, ArtikelNr, Year(VerkDatum)';
     EndSL;

     EXEC SQL  Close C1;

     //Konvertierung des SQL-SELECT-Statements und Zuweisung der SQLDA
     Exec SQL  Prepare DynC1
               into :SQLDA using System Names from :CmdSQL;;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung
        Return;
     EndIf;

     //Zuordnung SQLDA der Daten-Pointer zu Host-Strukturfeldern
     For Index = 1 to SQL_NUM;
         SQLVAR = SQL_VAR(Index);
         Select;
         When SQLName = 'KUNDENR';
              SQLData = %Addr(C1Kunde);
         When SQLName = 'ARTIKELNR';
              SQLData = %Addr(C1Artikel);
         When SQLName = 'JAHR';
              SQLData = %Addr(C1Jahr);
         When SQLName = 'GESAMT';
              SQLData = %Addr(C1Gesamt);
         Other;
              Clear SQLVAR;
         EndSL;
         SQLIND = %Addr(SQLCODE);                 //SQLCODE
         SQL_VAR(Index) = SQLVAR;
     EndFor;

     Exec SQL  Declare C1 Cursor For DynC1;

     EXEC SQL  Open  C1;
     If SQLCOD < *Zeros;
```

```
       //Fehlerbehandlung
       Return;
    EndIf;

   DoU 1=0;

       EXEC SQL Fetch Next From C1 into Descriptor :SQLDA;
       If SQLCODE = 100;
          Leave;
       ElseIf SQLCODE < *Zeros;
          //Fehlerbehandlung
          leave;
       EndIf;

       DsplyText = %Char(C1Jahr)    + ' / ' +  %Trim(C1Kunde)   + ' / ' +
                   %Trim(C1Artikel) + ' / ' +  %Char(C1Gesamt);
       Dsply DsplyText;
     EndDo;

     EXEC SQL  Close C1;
     If SQLCOD < *Zeros;
        //Fehlerbehandlung
        Return;
     EndIf;
     Return;
  /End-Free
P DspUmsKundeArtNrSortx...
P                 E
 ******* Quelle EMBDYNCDA1
```

Dynamisches SQL mit variabler SELECT-Liste

3.2.5.18 Verarbeitung von Result Sets aus Stored Procedures

Stored Procedures werden in Client-Server-Anwendungen gerne verwendet, um den Datenverkehr zwischen Client und Server auf ein Minimum zu reduzieren. Anstatt Befehle einzeln auszuführen oder Datensätze einzeln zu übertragen, werden diese in einem Programm (Stored Procedure) zusammengefasst, das mit Hilfe des SQL-Befehls CALL aufgerufen und ausgeführt wird.

Informationen können in Form von Ein-/Ausgabe-Parametern ausgetauscht werden.

Es besteht jedoch auch die Möglichkeit, eine unbestimmte Anzahl von Informationen/Datensätzen als Result-Set auszugeben.

Vor Release 7.1 war es für RPG, Cobol und die SQL-Programmiersprache nicht oder nur sehr aufwendig mit Hilfe von SQL CLI (Common Level Interface) APIs möglich, Result-Sets zu verarbeiten, die aus externen oder SQL Stored Procedures ausgegeben wurden.

Mit Release 7.1 wurden neue SQL-Befehle implementiert, mit deren Hilfe von Stored Procedures ausgegebene Result-Sets verarbeitet werden können.

Für die Verarbeitung von Result-Sets sind die folgenden Schritte erforderlich:

- Definition von Result-Set Locator-Variablen (Datentyp RESULT_SET_LOCATOR)
- Aufruf der Stored Procedure mit dem SQL-Befehl CALL
- Zuordnen der Result-Set Locator-Variable zu den zurückgegebenen Result-Sets mit dem SQL-Befehl ASSOCIATE RESULT SET LOCATORS
- Definieren/Deklarieren und Öffnen der Cursor für die zugeordneten Result-Sets mit dem SQL-Befehl ALLOCATE LOCATOR
- Verarbeiten des/der Cursor(s) in einer Schleife mit dem SQL-Befehl FETCH
- Schließen des/der Cursor(s) mit dem SQL-Befehl CLOSE

3.2.5.18.1 Host-Variablen mit Datentyp RESULT_SET_LOCATOR

Um in SQL, RPG oder Cobol ein aus einer Stored Procedure ausgegebenes Result-Set verarbeiten zu können, muss zunächst ein Handle definiert werden, über das das Result-Set innerhalb der Quelle eindeutig identifiziert werden kann. Dafür muss eine Variable mit dem Datentyp RESULT_SET_LOCATOR definiert werden. Intern wird für diesen Datentypen eine Integer-Variable bereitgestellt.

Weder für RPG noch für Cobol ist ein nativer Datentyp RESULT_SET_LOCATOR vorgesehen. Dennoch können Host-Variablen in RPG mit Hilfe des Schlüsselworts SQLTYPE als RESULT_SET_LOCATORs definiert werden.

Das folgende Beispiel zeigt die Definition von Result_Set_Locator-Variablen mit RPG:

```
D RSLCsrC01 S SQLTYPE(Result_Set_Locator)
D RSLCsrC02 S SQLTYPE(Result_Set_Locator)
D RSLCsrC03 S SQLTYPE(Result_Set_Locator)
```

Definition von Result_Set_Locator-Variablen mit RPG

Auch in Cobol können Result_Set_Locator-Variablen mit Hilfe des Schlüsselworts SQL TYPE definiert werden, wie das folgende Beispiel zeigt.

```
01 RSLCsrC01 SQL TYPE IS RESULT_SET_LOCATOR.
01 RSLCsrC02 SQL TYPE IS RESULT_SET_LOCATOR.
01 RSLCsrC02 SQL TYPE IS RESULT_SET_LOCATOR.
```

Definition Result_Set_Locator mit Cobol

3.2.5.18.2 SQL-Befehl CALL – Aufruf von Stored Procedures

Beim Aufruf von Stored Procedures müssen lediglich die Input- und Output-Parameter übergeben werden, unabhängig davon, ob die Stored Procedure Result-Sets zurückgibt oder nicht.

Ob und wie viele Result-Sets von einer Stored Procedure zurückgegeben werden, kann entweder über die SQL Communication Area oder die Diagnostics Area wie folgt ermittelt werden:

SQLCA
: In der SQLCA werden die Informationen über ausgegebene Result-Sets als Warnungen in SQLCOD/SQLCODE bzw. SQLSTT/SQLSTATE ausgegeben.

  Werden ein oder mehrere Result-Sets ausgegeben, werden der SQLCOD/SQLCODE +466 und der SQLSTT/SQLSTATE 0100C ausgegeben.

  Über die SQLCA kann nur ermittelt werden, ob Result-Sets ausgegeben wurden. Es ist jedoch nicht möglich, die Anzahl der ausgegebenen Result-Sets über die SQLCA zu bestimmen.

Diagnostics Area
: Über den SQL-Befehl GET DIAGNOSTICS können nicht nur die gleichen Informationen wie über die SQLCA ermittelt werden, sondern es werden auch zusätzliche oder detailliertere Informationen bereitgestellt.

So kann über das Schlüsselwort DB2_NUMBER_RESULT_SETS die Anzahl der aus einer aufgerufenen Stored Procedure zurückgegebenen Result-Sets ermittelt werden.

Im folgenden Beispiel werden die Stored Procedures HSINFO und HSMULTRS mit Hilfe des SQL-Befehls CALL aufgerufen.

Nach Aufruf der Stored Procedure HSINFO wird zunächst der SQLCOD/SQLCODE dahingehend geprüft, ob die Stored Procedure ordnungsgemäß ausgeführt wurde. Ist das der Fall, so wird geprüft, ob Result-Sets ausgegeben wurden. Die Anzahl der Result-Sets kann über die SQLCA nicht ermittelt werden.

Wurde die Stored Procedure HSMULTRS ordnungsgemäß ausgeführt, d. h. wurde ein positiver SQLCODE ausgegeben, wird mit Hilfe des SQL-Statements GET DIAGNOSTICS die Anzahl der ausgegebenen Result-Sets ermittelt.

```
D NbrResSets S 10I 0
*----------------------------------------------------------------
/Free
  //1. Prüfen Result-Sets aus Stored Procedures ausgegeben mit SQLCA
  Exec SQL Call HSINFO(:Parm1, :Parm2);

  If SQLCODE < *Zeros;
    //Fehler beim Ausführen der Stored Procedure HSINFO
  ElseIf SQLCODE = 466;
    //Result Set(s) Returned
  EndIf;

  //2. Ermitteln Anzahl Result-Sets mit Get Diagnostics
  Exec SQL Call HSMultRS();

  If SQLCODE < *Zeros;
    //Fehler beim Ausführen der Stored Procedure HSMULTRS
  Else;
      Exec SQL Get Diagnostics :NbrResSets = DB2_NUMBER_RESULT_SETS;
      If NbrResultSets > *Zeros;
      //Result Set(s) Returned
    EndIf;
  EndIf;
  ...
/End-Free
**************** Quelle EMBRESSET1
```

Aufruf Stored Procedure – Prüfen/Ermitteln der Anzahl Result-Sets

3.2.5.18.3 SQL-Befehl ASSOCIATE LOCATORS – Zuordnen von Result-Sets

Nachdem nun die Stored Procedure ausgeführt und die Anzahl Result-Sets ermittelt wurde, muss jedem Result-Set ein Result-Set-Locator/Handle zugeordnet werden.

Die Result-Set Locator-Variablen, die zugeordnet werden sollen, müssen in den D-Bestimmungen mit Hilfe des Schlüsselworts SQLTYPE als RESULT_SET_LOCATOR definiert worden sein.

Die Zuordnung der Result-Set-Locator-Variablen erfolgt mit dem SQL-Befehl ASSOCIATE RESULT SET LOCATORS. Werden mehrere Result-Sets aus einer Stored Procedure ausgegeben, müssen alle Result-Set-Locators im gleichen ASSOCIATE RESULT SET LOCATORS-Statement zugewiesen werden. Dabei wird das erste Result-Set der ersten Result-Set-Locator-Variablen zugeordnet, das zweite Result-Set dem zweiten Locator usw. Wird das Statement ASSOCIATE RESULT SET LOCATORS mehrfach für die gleiche Stored Procedure aufgerufen, beginnt die Zuweisung immer mit dem ersten Result-Set.

Da Stored Procedures überladen werden können – d. h. in einer Prozedur können mehrere Prozeduren mit dem gleichen Namen, jedoch mit einer unterschiedlichen Anzahl Parametern vorhanden sein – müssen entweder die Datentypen der Parameter oder der spezifische Name der Prozedur angegeben werden.

```
/Free
  //Syntax - Prozedur + Datentypen der Parameter
  Exec SQL Associate Result Set Locators (:HostRSLoc1, :HostRSLoc2, ...
                                          :HostRSLocN)
           With Procedure ProcName (ParmType1, ParmType2, ... ParmTypeX);
  //Syntax - Spezifischer Prozedurname
  Exec SQL Associate Result Set Locators (:HostRSLoc1, :HostRSLoc2, ...
                                          :HostRSLocN)
           With Specific Procedure ProcName;
```

Syntax ASSOCIATE LOCATORS – Zuordnen von Result-Set-Locators

Im folgenden Beispiel werden die RESULT_SET_LOCATOR-Variablen RSLCsrC01 bis RSLCsrC04 den Result Sets 1–4 aus der Stored Procedure HSMULTRS zugeordnet. Da diese Stored Procedure ohne Parameter aufgerufen und auch nicht der spezifische Name verwendet wird, muss nach dem Stored Procedure-Name eine leere Klammer angegeben werden.

Die Result-Set-Locator-Variable RSLCsrC05 wird dem ersten Result-Set in der Stored Procedure mit dem spezifischen Namen HSINF00003 zugewiesen.

Nach der Zuweisung der Result-Sets wird jeweils über den SQLCODE geprüft, ob das SQL-Statement erfolgreich ausgeführt wurde.

```
* Define Result Set Locators
D RSLCsrC01       S                   SQLTYPE(Result_Set_Locator)
D RSLCsrC02       S                   SQLTYPE(Result_Set_Locator)
D RSLCsrC03       S                   SQLTYPE(Result_Set_Locator)
D RSLCsrC04       S                   SQLTYPE(Result_Set_Locator)
D RSLCsrC05       S                   SQLTYPE(Result_Set_Locator)
 *---------------------------------------------------------------------
 /Free
    //Call Procedures
    //Associate Result-Set-Locators
    Exec SQL  Associate Result Set Locators (:RSLCsrC01, :RSLCsrC02,
                                             :RSLCsrC03, :RSLCsrC04);
                  with Procedure HSMultRS();
    If SQLCODE < *Zeros;
       //Fehlerbehandlung
    EndIf;

    Exec SQL  Associate Result Set Locators (:RSLCsrC05)
                  with Specific Procedure HSINF00003;
    If SQLCODE < *Zeros;
       //Fehlerbehandlung
    EndIf;

 /End-Free
```

SQL-Befehl ASSOCIATE RESULT SET LOCATOR – Zuordnen von Result Sets

3.2.5.18.4 SQL-Befehl ALLOCATE CURSOR – Result-Set öffnen

Im nächsten Schritt muss ein Cursor für die zugeordneten Result-Sets definiert und geöffnet werden. Dies erfolgt über den Befehl ALLOCATE CURSOR.

Der Cursor-Name ist frei wählbar, muss jedoch innerhalb der Quelle, in der das SQL-Statement verwendet wird, eindeutig sein. Dabei spielt es keine Rolle, ob innerhalb der Quelle diverse unabhängige Prozeduren codiert wurden.

Es darf jeweils nur ein Cursor einer Result-Set-Locator-Variablen zugeordnet werden.

```
/Free
   //Syntax
   Exec SQL Allocate CsrName Cursor For Result Set :HostRSLoc;
```

Syntax ALLOCATE CURSOR – Cursor für Result-Set zuweisen und öffnen

Ein mit ALLOCATE CURSOR zugeordnetes Result-Set wurde bereits in der aufgerufenen Stored Procedure geöffnet, d. h. die Ausführung des SQL-Statements OPEN CURSOR ist innerhalb des embedded SQL Programms weder notwendig noch erlaubt. Das ALLOCATE CURSOR-Statement kann als Kombination der SQL-Statements DECLARE CURSOR und OPEN verstanden werden.

Ein mit ALLOCATE CURSOR definierter und geöffneter Cursor kann mit dem SQL-Statement CLOSE geschlossen werden. Sofern die Stored Procedure innerhalb einer Prozedur mehrfach aufgerufen werden soll, müssen die den Result-Sets zugeordneten Cursor vor dem erneuten Aufruf geschlossen sein, anderenfalls könnte es zu Fehlern beim Aufruf der Stored Procedure kommen.

Anmerkung:

Die einfachste Methode, um solche Probleme zu vermeiden, ist, die Cursor vor dem Aufruf der Stored Procedure – d. h. vor Ausführung des CALL-Statements – zu schließen.

Sofern die Cursor geschlossen waren, kann die entsprechende Fehlermeldung ignoriert werden. Sofern die Cursor noch geöffnet waren, sind sie nun geschlossen.

Das folgende Beispiel zeigt den Aufruf der Prozedur MyProc. Nach dem Aufruf der Prozedur werden die ersten fünf Result-Sets Result-Set-Locator-Variablen zugeordnet.

3.2.5
Seite 157

Im folgenden Beispiel wird für die über Result-Set-Locator-Variablen zugeordneten Result-Sets jeweils ein Cursor definiert und geöffnet.

```
/Free
   //Ausführen der Stored Procedure(s)
     Exec SQL  Call MyProc();
     ...

   //Zuordnen der Result-Set(s) zu Result-Set-Locator-Variablen
     Exec SQL  Associate Result Set Locators (:RSLCsrC01, :RSLCsrC02,
                                              :RSLCsrC03, :RSLCsrC04,
                                              :RSLCsrC05)
                    With Specific Procedure MyProc;
     ...

   //Definieren/Öffnen Cursor für ausgegebene Result-Sets
     Exec SQL  Allocate CsrC01 Cursor For Result Set :RSLcsrC01;
     If SQLCODE < *Zeros;
        //Fehlerbehandlung
     EndIf;

     Exec SQL  Allocate CsrC02 Cursor For Result Set :RLSCsrC02;
     ...
     Exec SQL  Allocate CsrC03 Cursor For Result Set :RLSCsrC03;
     ...
     Exec SQL  Allocate CsrC04 Cursor for Result Set :RLSCsrC04;
     ...
     Exec SQL  Allocate CsrC05 Cursor For Result Set :RLSCsrC05;
     ...
```

SQL-Befehl ALLOCATE CURSOR – Definieren/Öffnen Cursor – Beispiel

Ergänzung 21/2012

3.2.5.18.5 SQL-Befehl FETCH – Verarbeiten von Result-Sets aus Procedures

Cursor, die mit ALLOCATE CURSOR definiert und geöffnet wurden, können genau wie Cursor, die mit dem SQL-Befehl DECLARE CURSOR definiert wurden, verarbeitet werden.

Mit dem Befehl FETCH können die im Result-Set ausgegebenen Informationen Zeile für Zeile verarbeitet werden. Das Ergebnis kann in (Host-)Variablen oder in Host-Strukturen ausgegeben werden.

Anstatt die Datensätze einzeln zu verarbeiten, können die Datensätze auch blockweise empfangen und in Array- oder Mehrfach-Datenstrukturen ausgegeben werden (Multiple Row Fetch).

Sofern der Cursor für das Result-Set im DECLARE CURSOR-Statement innerhalb der aufgerufenen Stored Procedure als serieller Cursor definiert wurde, können die Datensätze des Result-Sets in der rufenden Prozedur nur sequenziell verarbeitet werden (FETCH NEXT).

Wurde der Cursor jedoch als Scroll-Cursor definiert, können in der rufenden Prozedur die Datensätze des Result-Sets vorwärts und rückwärts gelesen werden. Ebenso ist es möglich, an den Anfang, an das Ende oder innerhalb des Result-Sets zu positionieren.

```
/Free
  //Syntax - serieller Cursor
  Exec SQL Fetch Next From MyCursor into :MyHostVar1,
                                         :MyHostVar2, :MyIndVar2
                                         :MyHostDS1,
                                         :MyHostDS2,  :MyIndArrDs2;

  //Syntax - Scroll-Cursor
    Exec SQL  FETCH Next (Prior/First/Last/Before/After/Current/Relative n)
                    FROM MyCursor
                    INTO :MyHostVar1, :MyHostVar2 :MyIndVar2,
                         :MyHostDS1,  :MyHostDS2  :MyIndArrDS2, ...;
```

Syntax des SQL-Befehls FETCH zur Verarbeitung von Result-Sets

3.2.5.18.6 SQL-Befehl CLOSE CURSOR für Result-Set

Wie bei der normalen Cursor-Verarbeitung sollten auch die Cursor zur Verarbeitung von Result-Sets aus Stored Procedures am Ende der Verarbeitung mit dem SQL-Befehl CLOSE geschlossen werden.

Ob am Ende des Moduls ein Hard Close (ODPs = Open Data Paths werden gelöscht) erfolgt, hängt von den Angaben in der Option CLOSQLCSR im Kompilierungsbefehl oder in einer SET OPTION-Anweisung ab.

Bei CLOSQLCSR = *ENDMOD werden die ODPs (unabhängig davon, ob sie wiederverwendbar sind oder nicht) bei Modulende gelöscht. Bei Angabe von CLOSQLCSR = *ENDACTGRP bleiben die wiederverwendbaren ODPs so lange geöffnet, bis die Aktivierungsgruppe beendet wird.

Cursor müssen vor dem erneuten Aufruf der Stored Procedure, die das Result-Set ausgibt, geschlossen sein.

Anmerkung:

Um Probleme, die durch geöffnete und nicht geschlossene Cursor verursacht werden können, zu vermeiden, sollten die Cursor unmittelbar vor Ausführung des SQL-CALL-Befehls geschlossen werden.

Sofern die Cursor geschlossen waren, kann die entsprechende Fehlermeldung ignoriert werden. Sofern die Cursor noch geöffnet waren, sind sie nun geschlossen.

3.2.5.18.7 Verarbeiten von Result-Sets aus Stored Procedures – Beispiel

Um zu demonstrieren, wie aus einer Stored Procedure ausgegebene Result-Sets verarbeitet werden, wird zunächst die Stored Procedure erstellt, die Result-Sets ausgibt.

Das folgende Beispiel zeigt das SQL-Script der Stored Procedure „GetCompany“, die einen einzigen alphanumerischen Parameter mit variabler Länge empfängt und ein Result-Set ausgibt.

Innerhalb der Stored Procedure wird ein serieller Cursor definiert, über den alle Datensätze aus dem Adress-Stamm ermittelt werden, die entweder in Name1 oder Name2 die übergebene Zeichenkombination enthalten.

Das Result-Set wird dadurch ausgegeben, dass der Cursor am Ende der Prozedur geöffnet bleibt. Da der Cursor am Ende der Prozedur nicht geschlossen werden darf und da auch nicht sichergestellt werden kann, dass die rufende Prozedur den Cursor schließt, wird der Cursor vor dem Öffnen zunächst geschlossen. Konnte der Cursor nicht geschlossen werden, da er nicht geöffnet war, wird der Fehler SQLSTATE 24501 ausgelöst, durch den der definierte CONTINUE HANDLER aktiviert wird. Nachdem ein Continue Handler abgearbeitet ist, wird die Verarbeitung mit dem nächsten Statement fortgesetzt.

3.2.5
Seite 161

Nach Ausführung des CLOSE-Statements wird der Cursor geöffnet, wodurch das Result-Set erstellt und die Prozedur beendet wird.

```
Create Procedure MySchema/GetCompany (IN ParName VarChar(50))
       Dynamic Result Sets 1
       Language SQL
       Not Deterministic
       Reads SQL Data
       Called On NULL Input

Begin Not Atomic
   Declare VarNotOpen SmallInt;
   Declare CsrNotOpen Condition For '24501';

   Declare CsrC01 Cursor For
      Select KundeNr, Name1F, Strasse, Land, PLZ, Ort
         From AdresseX
         Where    Upper(Name1F) like '%' concat ParName concat '%'
               or Upper(Name2F) like '%' concat ParName concat '%';

   Declare Continue Handler for CsrNotOpen Set VarNotOpen = 1;

   Set ParName = Upper(Trim(ParName));

   Close CsrC01;   Open  CsrC01;
End;
--********* Quelle GETCOMPANY
```

SQL Stored Procedure mit Ausgabe eines Result-Sets

3.2.5
Seite 162

Im folgenden Beispiel wird die zuvor beschriebene SQL Stored Procedure „GetCompany“ ausgeführt und das ausgegebene Result-Set verarbeitet.

Vor Aufruf der Stored Procedure wird zunächst der Cursor geschlossen, sofern er geöffnet war.

Die Stored Procedure wird mit dem Parameter „ParSearch“, der den Wert „GmbH & Co“ enthält, aufgerufen. Nach dem Aufruf wird über SQLCODE geprüft, ob die Stored Procedure ordnungsgemäß verarbeitet wurde.

Das aus der Stored Procedure „GetCompany“ ausgegebene Result-Set wird in dem SQL-Statement ASSOCIATE RESULT SET LOCATOR dem Result-Set-Locator RSLCsrC01 zugeordnet.

Über den SQL-Befehl ALLOCATE CURSOR wird für das Result-Set ein Cursor definiert und geöffnet.

In einer Verarbeitungsschleife werden alle in dem Result-Set enthaltenen Datensätze einzeln empfangen und in die Host-Struktur DSCsrC01 übertragen. Nach Ausführung des FETCH-Statements wird geprüft, ob das Ende des Result-Sets (SQLCODE 100) erreicht wurde oder ob ein Fehler aufgetreten ist. In beiden Fällen wird die Verarbeitungsschleife verlassen.

Sofern ein Datensatz eingelesen werden konnte, werden die Firmen-Nr. und der Firmen-Name in einem String aufbereitet und mit Hilfe des OpCodes DSPLY ausgegeben.

Nachdem die Verarbeitungsschleife verlassen wurde, wird der Cursor geschlossen und das Programm beendet.

```
D Text            S             50A

D ParSearch       S             50A   Varying inz('GmbH & Co')

D RSLCsrC01       S                   SQLType(Result_Set_Locator)

D DSCsrC01        DS                  Qualified
D   CompanyNo                   15A
D   Company                     35A
D   Street                      35A
D   Country                      4A
D   ZipCode                      9A
D   City                        35A
 *-----------------------------------------------------------------
 /Free
    *INLR = *On;

    Exec SQL Close CsrC01;

    Exec SQL Call GetCompany(:ParSearch);
    If SQLCODE < *Zeros;                  //Fehler
       Return;
    EndIf;

    Exec SQL Associate Result Set Locator (:RSLCsrC01)
                  with Procedure GetCompany(VarChar());
    If SQLCODE < *Zeros;                  //Fehler
       Return;
    EndIf;

    Exec SQL Allocate CsrC01 Cursor For Result Set :RSLCsrC01;
    If SQLCODE < *Zeros;
       Return;
    EndIf;

    DoU 1=0;
        Exec SQL  Fetch Next From CsrC01 into :DSCsrC01;
        If SQLCODE < *Zeros or SQLCODE = 100;
           Leave;
        EndIf;

        Text = %Trim(DSCsrC01.CompanyNo) +' '+
```

```
              %Trim(DSCsrC01.Company);
         Dsply Text;                          EndDo;
                                                  Exec SQL Close CsrC01;
     If SQLCODE < *Zeros;              //Fehler
        Return;
     EndIf;

     Return;
/End-Free
************* EMBRESSET2
```

Verarbeiten eines Result-Sets mit embedded SQL

3.2.5.19 Diagnostics Area – SQL-Befehl GET DIAGNOSTICS

Der SQL-Befehl GET DIAGNOSTICS wurde mit Release V5R3M0 eingeführt und liefert Informationen über das zuletzt ausgeführte SQL-Statement. Diese Informationen werden mit Hilfe von Schlüsselwörtern (und dem SQL-Befehl GET DIAGNOSTICS) aus der Diagnostics Area ermittelt und in bereitgestellte Host-Variablen ausgegeben. In einem GET DIAGNOSTICS-Statement können gleichzeitig mehrere Informationen ausgegeben werden.

Neben den Informationen, die auch über die SQLCA bereitgestellt werden, liefert die Descriptor Area weitere oder detailliertere Informationen. So kann z. B. aus der SQLCA nur der Fehlercode ermittelt werden, der Nachrichtentext muss mühsam aus der Message-File QSQLMSGF ermittelt werden, während der Fehlertext über GET DIAGNOSTICS direkt ausgegeben werden kann.

Bei der Ausführung des GET DIAGNOSTICS-Statements wird die SQLCA nicht aktualisiert. Wird nach Ausführung des GET DIAGNOSTICS-Statements der Inhalt von SQLCODE oder SQLSTATE geprüft, so handelt es sich hierbei um die Werte, die durch das SQL-Statement, das vor GET DIAGNOSTICS ausgeführt wurde, gesetzt wurden.

Die Informationen, die über die Descriptor Area bereitgestellt werden, können in drei Blöcke unterteilt werden:

- Statement-Informationen (statement information):
  Zu diesen Informationen zählen u. a. die Anzahl der geänderten oder gelöschten Datensätze, die Anzahl der zurückgegebenen Result-Sets oder ob bei einem Multiple Row Fetch der letzte Datensatz ausgegeben wurde.
- Verbindungsinformationen (connection information)
  Zu diesen Informationen zählen u. a. die Art sowie der Status der Datenbankverbindung
- Bedingungsinformationen (condition information)
  Hier werden detailliertere Informationen über fehlerhaft ausgeführte SQL-Statements ausgegeben.
  Zu diesen Informationen zählen u. a. auch der Fehlertext, der Name der fehlerhaft ausgeführten Routine (Stored Procedure oder User Defined Function), der Name des fehlerhaft ausgeführten Triggers oder der Name der referenziellen Integrität, die die Integritätsverletzung verursacht hat.

3.2.5.19.1 Ermitteln von Statement- und Connection-Informationen

Die Syntax zum Ermitteln von Statement- und Connection-Informationen ist identisch.

```
//SQL-Statement, das geprüft werden soll
Exec SQL ... ;

//Syntax für Statement- und Connection-Informationen
Exec SQL  GET DIAGNOSTICS
             KeyWord1 = :HostVar1, KeyWord2 = :HostVar2, ...
             KeyWordN = :HostVarN;
```

Syntax GET DIAGNOSTICS für Statement- und Connection-Informationen

Die folgende Tabelle gibt einen Überblick über die wichtigsten Schlüsselwörter der Statement-Informationen:

Statement Informationen			
Schlüssel-Wort	Definition		Bemerkung
	SQL Variable	RPG Variable	
DB2_LAST_ROW	Integer	10I 0	Sofern zuvor ein Mulitple Row Fetch-Statement ausgeführt wurde, wird, sofern der letzte Satz eingelesen wurde, +100 ausgegeben.
DB2_NUMBER_RESULT_SETS	Integer	10I 0	Sofern es sich bei dem vorherigen SQL-Statement um einen CALL Stored Procedure-Befehl handelte, wird die Anzahl der zurückgegbenen Result-Sets ausgegeben
DB2_NUMBER_ROWS	Dec(31, 0)	31P 0	Sofern es sich bei dem vorherigen SQL-Statement entweder um ein OPEN oder FETCH-Statement handelt, wird die Anzahl der Zeilen im Result-Set zurückgegeben. Bei sensitiven Cursorn handelt es sich dabei lediglich um einen Nährungswert.
ROW_COUNT	Dec(31, 0)	31P 0	Sofern es sich bei dem vorherigen SQL-Statement um ein DELETE-, UPDATE- oder INSERT-Statement handelt, wird die Anzahl der gelöschten, geänderten oder eingefügten Datensätze ausgegeben. Sofern zuvor ein Mulitple Row Fetch-Statement ausgeführt wurde, wird die Anzahl der im Block eingelesenen Zeilen ausgegeben.

Diagnostics Area – Statement-Informationen

Weitere Schlüsselwörter sind im IBM i Information Center unter folgendem Link dokumentiert:

http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Fdb2%2Frbafzgetdiag.htm

Im folgenden Beispiel wird nach einem Multiple Row Fetch die Anzahl der empfangenen Zeilen (Schlüsselwort ROW_COUNT) ermittelt. Weiterhin wird geprüft, ob die letzte Zeile in den empfangenen Zeilen enthalten war (Schlüsselwort DB2_LAST_ROW).

Basierend auf diesen Informationen werden die empfangenen Datensätze verarbeitet. Anschließend wird entweder der nächste Block Zeilen eingelesen oder die Verarbeitungsschleife verlassen.

Nach Ausführung des SQL-Statements wird der SQLCODE des FETCH-Statements geprüft. (SQLCODE und SQLSTATE bleiben bei Ausführung eines GET DIAGNOSTICS-Statements unverändert.

```
D RtnNbrRows      S             10I 0
D RtnLastRow      S             10I 0

D ParNbrRowsPage  S             10I 0  inz(100)

D DSCsrC1         DS                   Qualified Dim(100)
D*** Unterfelder
 *----------------------------------------------------------------------
 /Free

    Exec SQL  Declare CsrC1 Cursor For
                  Select ... ;

    Exec SQL  Open CsrC1;
    ...
    DoU 1=0;
        Exec SQL Fetch Next From CsrC1
                       For :ParRowsPage Rows into :DSCsrC1;
        Exec SQL Get Diagnostics :RtnNbrRows = ROW_COUNT,
                                 :RtnLastRow = DB2_LAST_ROW;
        If    RtnNbrRows = *Zeros
           or SQLCODE = 100;
           Leave;
        EndIf;

        For Index = 1 to RtnNbrRows;
            //Verarbeitung der Datensätze
        EndFor;

        If    RtnNbrRows < %Elem(DSCsrC1)
           Or RtnNbrRows = %Elem(DSCsrC1 and RtnLastRow = 100;
           //Alle Datensätze verarbeitet
           Leave;
        EndIf;

    EndDo;
 /End-Free
 ************ Quelle EMBGETDIA
```

Diagnostics Area – Verarbeitung von Statement-Informationen – Beispiel

3.2.5.19.2 Ermitteln von Bedingungsinformationen

Für Bedingungsinformationen muss die Nummer der Bedingung angegeben werden. Jede Fehlernachricht, die aus einem SQL-Statement resultiert, wird einer Nr. zugeordnet. Bedingung 1 korrespondiert mit der Fehlerursache aus dem vorangegangenen SQL-Statement. In der Regel genügt es, Informationen für die erste Bedingung zu verarbeiten.

```
/Free
   //SQL-Statement, das geprüft werden soll
   Exec SQL ... ;

   //Syntax für Bedingungsinformationen
   Exec SQL  GET DIAGNOSTICS Condition 1
                KeyWord1 = :HostVar1, KeyWord2 = :HostVar2, ...
                KeyWordN = :HostVarN;
```

Diagnostics Area – Bedingungsinformationen – Syntax

Die am häufigsten benötigten Informationen beziehen sich auf die Fehlermeldung und können über die folgenden Schlüsselwörter ermittelt werden:

Bedingungsinformationen			
Schlüssel-Wort	Definition		Bemerkung
	SQL Variable	RPG Variable	
DB2_MESSAGE_ID	Char(10)	10A	Nachrichten-Nr. des Fehlertextes in Message-File QSQLMSG
DB2_MESSAGE_ID1	VarChar(7)	7A Varying	CPF-Message-Id des Fehlertextes
DB2_MESSAGE_ID2	VarChar(7)	7A Varying	CPD-Message-Id des Fehlertextes
MESSAGE_LENGTH	Integer	10I 0	Anzahl der Zeichen des Nachrichtentextes
MESSAGE_TEXT	VarChar(32740)	32749A Varying	Komplett aufbereiteter Fehlertext (max. 32740 Zeichen)
			Variable Nachrichteninformationen sind integriert

Diagnostics Area – Bedingungsinformationen

Weitere Schlüsselwörter sind im IBM i System i Information Center unter folgendem Link dokumentiert:

http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Fdb2%2Frbafzgetdiag.htm

Im folgenden Beispiel werden – sofern ein Delete-Statement nicht ordnungsgemäß ausgeführt werden konnte (SQLCODE < *Zeros) – der zugehörige Fehlertext (Schlüsselwort MESSAGE_TEXT) sowie die Länge der Fehlernachricht (Schlüsselwort MESSAGE_LENGTH) über die Diagnostics Area ermittelt.

3.2.5
Seite 169

Sofern Datensätze gelöscht wurden (SQLCODE <> 100), wird die Anzahl der gelöschten Datensätze (Schlüsselwort ROW_COUNT) über die Diagnostics Area festgestellt.

```
D LenErrorText    S             10I 0
D ErrorText       S          32740A    Varying
D NbrRowsDeleted  S             10I 0
 *----------------------------------------------------------------
/Free
   Exec SQL  Delete From MyTable
                Where ...;

     If SQLCODE < *Zeros;
        Exec SQL Get Diagnostics Condition 1
                    :LenErrorText = MESSAGE_LENGTH,
                    :Error_Text   = MESSAGE_TEXT;
     ElseIf SQLCODE <> 100;
        Exec SQL Get Diagnostics :NbrRowsDeleted = ROW_COUNT;
     EndIf;
```

Diagnostics Area – Bedingungsinformationen – Beispiel

3.2.5
Seite 170

3.2.6 RPG und Daten

Die meisten Programme haben das Ziel, Daten zu verarbeiten. Es handelt sich hierbei um Daten, die im Falle des System i über die relationale Datenbank DB2/UDB zur Verfügung stehen. Dazu gehört auch der Einsatz des IFS (Integrated File System), was jedoch hier nicht weiter berücksichtigt werden soll. IBM macht es System-i-Betreibern nicht unbedingt leicht, denn es gibt im Sinne des Datenbankdesigns heute mehrere Varianten, wie Daten gespeichert werden können. Es handelt sich um folgende:

Im klassischen ISAM-Format

- Bibliotheken, die Dateien beinhalten:
  - Physische Dateien (Datencontainer)
  - Logische Dateien (logische Sichten zu einer physischen Datei)

Im von IBM empfohlenen SQL-Format

- Schema. Darin sind enthalten:
  - SQL-Tabellen
  - SQL-Indizes und -Ansichten

Für RPG-Entwickler verhalten sich die beiden Varianten nahezu identisch. Das bedeutet, dass mit den in RPG bereitgestellten Funktionen die klassischen Datenbankelemente (physische und logische Dateien) in derselben Art und Weise verarbeitet werden können wie die modernen SQL-Derivate (Tabellen, Indizes und Ansichten).

Das Genannte soll dem groben Verständnis der Inhalte der DB2/UDB dienen. Details dazu finden Sie in anderen Kapiteln, die sich eingehend mit SQL und der Datenbank beschäftigen.

An dieser Stelle soll jedoch gezeigt werden, wie auf den Datenbankinhalt zugegriffen werden kann, um mit den darin befindlichen Daten arbeiten zu können.

Im Vergleich zu anderen Sprachen bewegt man sich mit RPG direkt auf dem System, auf dem sich auch die Datenbank selbst befindet. Das hat den Vorteil, dass zum Beispiel keine besonderen Treiber benötigt werden, mit denen eine Verbindung zur Datenbank hergestellt werden muss. Alles, was für einen Datenzugriff auf die DB2-Informationen mit RPG benötigt wird, ist im RPG implementiert. Natürlich können dabei auch weitere Funktionen des Betriebssystems IBM i und der Datenbank, wie zum Beispiel Berechtigungsprüfungen und die automatische Dateiverwaltung, genutzt werden.

Zunächst soll gezeigt werden, wie auf den Dateiinhalt mit RPG lesend zugegriffen werden kann.

3.2.6.1 Daten einer Datei mit RPG lesen

Der Zugriff auf den Inhalt einer DB2-Datei kann im RPG mit unterschiedlichen Anweisungen erfolgen. Welche Anweisung für den Einzelfall anzuwenden ist, ist meist durch die Beschaffenheit und die Organisation der Daten bestimmt. Besteht eine direkte Zugriffsmöglichkeit über Schlüsselbegriffe, dann kann mit diesen auf die gewünschten Informationen zielgerichtet zugegriffen werden. Anders stellt es sich dar, wenn kein direkter Zugriff zur Verfügung steht. In diesem Fall müssen alle Datensätze innerhalb einer Datei gelesen werden. Es handelt sich hierbei um sequenzielles Lesen, bei dem Satz für Satz verarbeitet wird.

Das Erlernen von RPG gestaltet sich insbesondere am Anfang als schwierig, was zum einen in den Spezifika des System i begründet ist, zum anderen aber auch in den vielen verschiedenen Möglichkeiten, die IBM den Anwendungsentwicklern zur Auswahl bereitstellt. Von jeher hat IBM System-i-Betreibern die Möglichkeit geboten, die neuesten Maschinen mit schon beinahe antiquierten Techniken betreiben zu können. So können zum Beispiel auch heute – fast 30 Jahre nach Einführung der AS/400 – die damals verfügbaren Urversionen von RPG verwendet werden. Im Falle des lesenden Zugriffs auf den Dateiinhalt werden Anweisungen für das Lesen in Verbindung mit Bezugszahlen genutzt. Diese Bezugszahlen oder auch Indikatoren liefern als Ergebnis ein „erfolgreich gelesen“ oder eben „nicht erfolgreich gelesen“. Obwohl IBM die Nutzung der modernen RPG-Techniken empfiehlt, haben Anwendungsentwickler hier letztlich die Qual der Wahl.

Da im Zuge der Nachfolge noch heute immer wieder auf alten RPG-Code gestoßen wird, soll neben dem Aufzeigen des Arbeitens mit Dateiinhalten auf die moderne Art auf die Historie verwiesen werden, damit Sie auch ältere RPG-Programme lesen und warten können.

Die Vielfalt ist allerdings derart hoch, dass nicht alle möglichen Optionen aufgezeigt werden können. Sollten Sie in einem Quellcode Dateinutzungen finden, die hier nicht beschrieben sind, dann empfiehlt es sich, zum Beispiel die RPG-Referenz der IBM als Quelle zu nutzen.

Nachfolgend sollen die ersten einfachen Beispiele für das Einlesen der Dateiinhalte vorgestellt werden.

Vor der Betrachtung der möglichen Lesezugriffe erfolgt eine kurze Beschreibung der Datei, die in den nachfolgenden Beispielen genutzt wird. Diese Datei beinhaltet Artikeldaten und verwendet als primären Zugriff die beiden Schlüsselfelder „Firma“ und „Artikelnummer“.

```
RPGREAD.RPGLE    DEMOART.PF
 Zeile 18      Spalte 36    Ersetzen
      .....A.............................Functions++++++++++++++++++
000100      ***********************************************************
000200      *
000300      *
000400      ***********************************************************
000500     A          R RDEMOART                  TEXT('BEISPIELDATEI 1')
000600     A            X1FIRM         3          COLHDG('Firma')
000700     A            X1TEIL        10          COLHDG('Teilenr')
000800     A            X1BEZE        50          COLHDG('Bezeichnung')
000900     A            X1MAGR         3          COLHDG('Materialgruppe')
001000     A            X1STAT         2          COLHDG('Status')
001100     A            X1KATA         3          COLHDG('Katalogkz.')
001200      *
001300      *   SCHLüSSELFELDER
001400      *
001500     A          K X1FIRM
001600     A          K X1TEIL
```

Beispieldatei DemoArt

In der Datei sind einige Beispieldaten enthalten:

X1FIRM	X1TEIL	X1BEZE	X1MAGR	X1STAT	X1KATA
01	4711	Heftklammern 1000 St}ck	01	A	001
01	5000	Kopierpapier	05	A	003
01	5500	Bleistift	06	A	004
01	5550	Toner	11	A	001
10	8000	Mercedes E220	AA	A	MB
10	8001	Audi A6	AA	A	MB

Beispieldatensätze in der Datei DemoArt

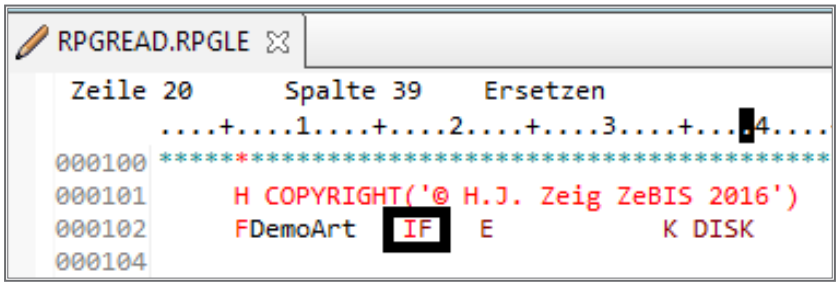
Wie es an den Beispieldatensätzen ersichtlich ist, sind in der Datei DemoArt zwei Firmen abgebildet. Die eine handelt mit Büroartikeln (Firma 01), die andere mit Gebraucht-Pkws (Firma 10).

Es kann bereits an den Symbolen in der RDI-Anzeige abgelesen werden, dass die beiden ersten Felder (x1firm und x1teil) die Schlüsselfelder sind.

Wie in jeder anderen Programmiersprache auch, muss in RPG zunächst einmal angegeben werden, welche Daten verarbeitet werden sollen. Dabei muss der Dateiname in den bereits bekannten Dateibestimmungen hinterlegt werden. Es ist bereits bekannt, dass sich die Daten in der DB2/UDB in den physischen Dateien bzw. in den Tabellen befinden. Der optimale Zugriff auf die Daten kann es erforderlich machen, dass mit Hilfe einer logischen Datei oder einer Sicht in RPG gearbeitet wird. Letztlich unterscheiden sich die beiden Dateivarianten physische und logische Datei nicht vom Codieren in den Dateibestimmungszeilen.

3.2.6
Seite 4

Der erste Schritt für das Lesen von Daten in einem RPG-Programm ist also das Codieren der Datei in den Dateibestimmungen. Da zunächst nur die Daten in einer Datei eingelesen werden sollen, wird die Datei als „Eingabedatei“ (Kennzeichen „I“ = Input in der Spalte „Dateiart“) definiert.



Datei zum Lesen (I) definiert

Anmerkung:

Um eine Datei lesen zu können, muss diese in den Dateibestimmungen mindestens als Eingabedatei (I) definiert sein. Das Einlesen ist jedoch auch für Dateien möglich, die zum Update (U) gekennzeichnet sind.

Beim nativen Lesen von Daten mit RPG (ohne SQL) ist diese Angabe ein Muss, da der Compiler die Dateibestimmungen dazu verwendet, um zum Beispiel alle in der Datei enthaltenen Felder dem RPG vorbereiten zu können.

Im nächsten Schritt muss der Zugriff auf den Dateiinhalt erfolgen.

Hier ist, wie zuvor erwähnt, die Wahl zu treffen, ob ein einzelner Datensatz gelesen werden soll und der Zugriff auf den oder die Datensätze mit Hilfe eines definierten Schlüssels möglich ist oder ob der Dateiinhalt sequenziell verarbeitet werden muss, damit die gewünschten Informationen bereitgestellt werden können.

In Abhängigkeit von der Verarbeitungsform muss bei Schlüsselzugriffen dafür gesorgt werden, dass die korrekten Daten eingelesen werden. Das bedeutet unter anderem, dass der Pointer so in der Datei positioniert wird, dass die gewünschten Daten eingelesen werden können.

3.2.6.1.1 READ – Lesen eines Datensatzes

Folgendes Beispiel verarbeitet die Dateiinhalte in sequenzieller Form, wozu der einfache Lesezugriff in Form der Anweisung „READ“ verwendet wird.

READ dient dazu, einen Datensatz in einer Datei einlesen zu können. Dabei wird kein gezielter Zugriff vorgenommen, sondern sequenziell gelesen – das bedeutet, dass in der Datei der nächste enthaltene Datensatz mit READ gelesen wird.

Im Beispiel werden die einzelnen Datensätze eingelesen, und die Artikelbezeichnung wird zur Anzeige gebracht.

RPGREAD.RPGLE

```
Zeile 22      Spalte 5     Einfügen
      ....+....1....+....2....+....3....+....4....+....5
000100 ******************************************************
000101      H COPYRIGHT('© H.J. Zeig ZeBIS 2016')
000102      FDemoArt   IF   E           K DISK
000104
000105       * Benannter Indikator
000106      d                 ds
000107      D Exit                    03     03N
000108
000109       ******************************************************
000110       /FREE
000111
000112        // Leseschleife
000113          Read RDemoArt;
000114          Dow not %eof;
000116           dsply X1Beze;
000117           Read RDemoArt;
000118          Enddo;
000119
000120          *InLr = *On;
000121
000122       /END-FREE
```

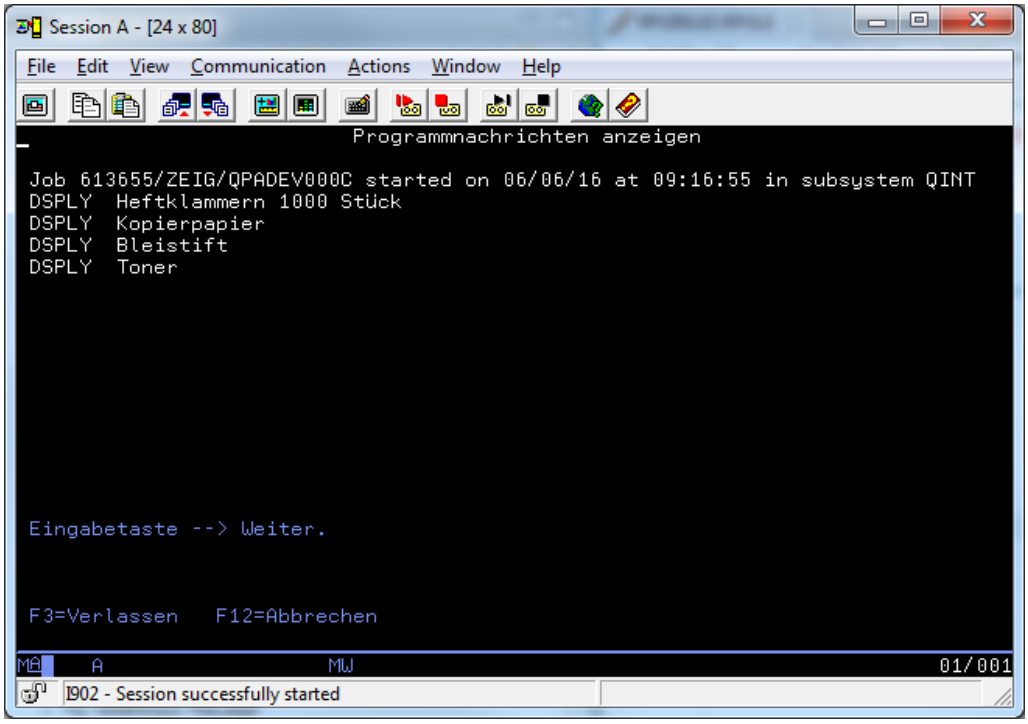
Leseschleife mit READ-Zugriff

Nun wurde der Lesezugriff mit Hilfe der READ-Anweisung durchgeführt und alle Datensätze in der Datei DemoArt verarbeitet. Die Datei hat zwei Schlüsselfelder – die Firma und die Teilenummer. Wenn diese Datei nun sequenziell mit READ verarbeitet wird, dann bestimmt die Schlüsseldefinition in der Datei die Reihenfolge der Verarbeitung. Dass der Inhalt des Feldes „Artikelbezeichnung“ (Feld X1BEZE) angezeigt wird, ist das Ergebnis des Lesezugriffs.

3.2.6
Seite 6

Die READ-Verarbeitung wird in einer DOW-Schleife eingebunden. Diese Schleife wird so lange verarbeitet, bis das Dateiende erreicht ist. Dieses wird im Beispiel mit Hilfe der Built-In-Function %EOF (End of File) geprüft.

Wird das Programm ausgeführt, dann ist als Beispiel die nachfolgende Anzeige zu sehen.



Ergebnisanzeige

READ lässt sich zum Verarbeiten einer beliebigen Zahl von Datensätzen ohne gezielten Zugriff verwenden. Allerdings eignet es sich nicht dazu, gezielt nach einem Datensatz zu suchen.

Dazu stehen in RPG andere Lesezugriffsoptionen zur Verfügung.

3.2.6.1.2 READE – Lesen eines Datensatzes

Die zielgerichtete Verarbeitung eines Datensatzes ist eine häufige Anforderung an RPG-Entwickler. Es werden die in der Datei definierten Schlüssel genutzt, und es wird mit den entsprechenden Schlüsselbegriffen auf den Dateiinhalt zugegriffen.

Mit Blick auf die Beispiel-Artikeldatei kann die Annahme getroffen werden, dass mit der Firma und der Artikelnummer genau auf einen Datensatz zugegriffen wird.

Dazu bedarf es in RPG folgenden Vorgehens:

1. Definieren der Datei in den Dateibestimmungen
2. Definieren eines Schlüssels in RPG (analog zu den Schlüsselfeldern in der Datei). Dabei kann auch ein Teilschlüssel verwendet werden. Das bedeutet, dass nicht zwingend alle Schlüsselfelder in RPG verwendet werden müssen. Wichtig ist jedoch, dass die Schlüsselfelder in der Reihenfolge genutzt werden, wie sie in der Datei definiert sind.
3. Positionieren in der Datei
4. Einlesen des Datensatzes (oder der Datensätze), die dem Schlüssel entsprechen

Nachdem das Definieren der Dateibestimmung bereits aufgezeigt worden ist, soll sich auf die Schlüsseldefinition konzentriert werden.

Wie bereits dargestellt, muss man sich mit der Schlüsseldefinition an die in der Datei definierten Schlüssel halten.

3.2.6
Seite 8

In dieser Datei sieht die Schlüsseldefinition folgendermaßen aus:

RPGREAD.RPGLE | DEMOART.PF

```
Zeile 1      Spalte 1     Ersetzen
      ...+A*..1....+....2....+....3....+....4....+....5....+....6....+....7.
000100      ********************************************************************
000200      *
000300      *
000400      ********************************************************************
000500     A          R RDEMOART                  TEXT('BEISPIELDATEI 1')
000600     A            X1FIRM         3         COLHDG('Firma')
000700     A            X1TEIL        10         COLHDG('Teilenr')
000800     A            X1BEZE        50         COLHDG('Bezeichnung')
000900     A            X1MAGR         3         COLHDG('Materialgruppe')
001000     A            X1STAT         2         COLHDG('Status')
001100     A            X1KATA         3         COLHDG('Katalogkz.')
001200      *
001300      *   SCHLüSSELFELDER
001400      *
001500     A          K X1FIRM
001600     A          K X1TEIL
001601
001602
```

Schlüsseldefinition in der Datei

Wie ersichtlich, werden in der Datei DemoArt zwei Schlüsselfelder genutzt:

- X1firm (Firma)
- X1teil (Teilenummer)

Damit wird der Dateiinhalt für die Verarbeitung im RPG-Programm zunächst nach dem ersten Schlüsselfeld (Firma) und danach nach dem nächsten Schlüsselfeld (Teilenummer) sortiert.

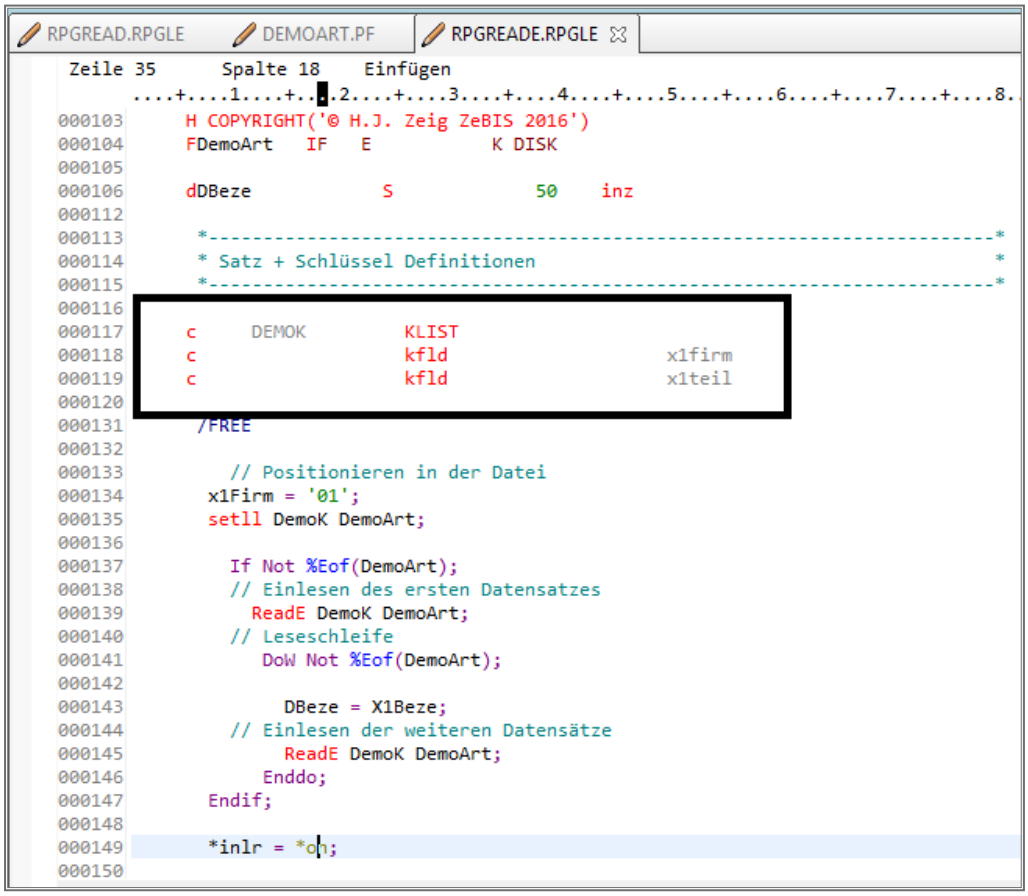
Dieser Schlüssel soll nun zusammen mit der Leseanweisung READE genutzt werden. Dazu gibt es in RPG verschiedene Optionen.

Als erste Option sei das Fix-Format genannt.

Im Fix-Format-RPG steht die Anweisung KLIST zur Verfügung. Diese Anweisung bedarf stets einer zweiten Anweisung kfld, die zwingend nach KLIST angegeben werden muss.

KLIST teilt dem RPG-Programm mit, dass eine Schlüsselliste definiert werden soll. Dieser muss ein eindeutiger Name zugeordnet werden, hier: DemoK.

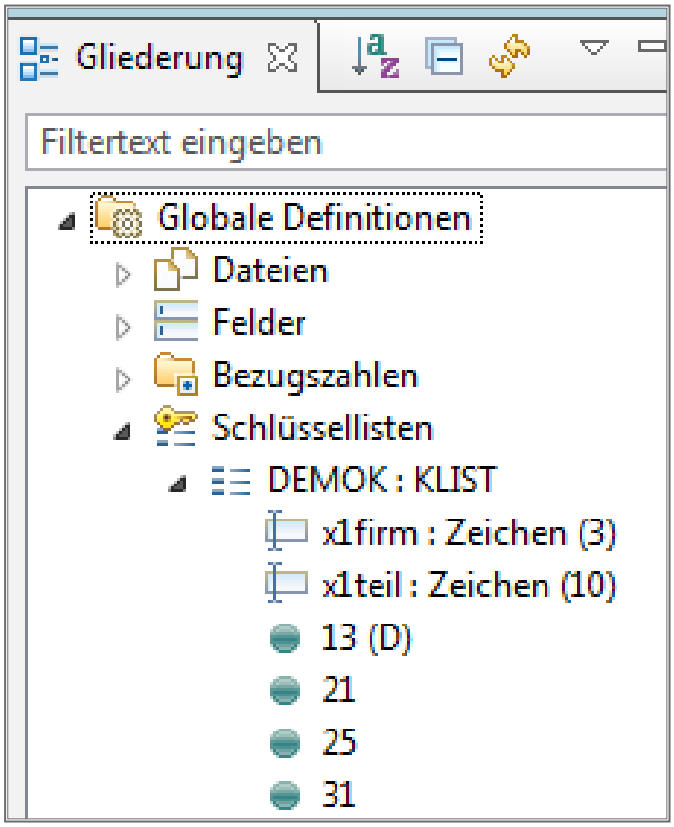
Diese Schlüsselliste wird mit den beiden Schlüsselfeldern definiert, die in der Datei existieren. Es werden dieselben Namen verwendet, wie sie auch in der Datei hinterlegt sind. Doch ist das kein Muss, denn es können auch andere Feldnamen genutzt werden. Wichtig ist nur, dass diese mit der Definition in der Datei in Bezug auf die Feldart und die Feldlänge übereinstimmen.



Schlüssellistendefinition mit KLIST

3.2.6
Seite 10

RDI zeigt in der Gliederungssicht weitere Details zur Schlüsselliste an. Hier können zum Beispiel die in der Liste enthaltenen Schlüsselfelder und deren Definitionen angezeigt werden.



Schlüssellistenanzeige in der RDI-Gliederungssicht

Nun wurden die Datei und die Schlüsselliste definiert. Es fehlt nur noch das Zuweisen eines Wertes in die gewünschten Schlüsselfelder.

Im Beispiel erfolgt dies, indem den Schlüsselfeldern die folgenden Werte zugewiesen werden:

- X1Firm = ‚01‘
- X1Teil = ‚5000‘

```
000133          // Schlüsselfelder Werte zuweisen
000134          x1Firm = '01';
000135          x1Teil = '5000';
```

Wertzuweisung im Schlüsselfeld

Achtung!

Hier ist eine Besonderheit bei der Verwendung von KLIST zu beachten. Zusammen mit KLIST wurde dem RPG-Programm mitgeteilt, dass die Schlüsselliste aus zwei Feldern besteht. Diese müssen zwingend zusammen genutzt werden, was bedeutet, dass beide Felder für das korrekte Verarbeiten mit einem Wert belegt werden müssen. Würde zum Beispiel nur einem der Schlüsselfelder (x1Firm) ein Wert zugewiesen werden, dann führe dies dazu, dass die Schlüsseldefinition nicht vollständig ist und damit die Verarbeitung nicht den gewünschten Effekt erzielt.

Das kann durchaus als Nachteil bei der Verwendung von KLIST angesehen werden.

Doch wie kann das umgangen werden, bzw. wie kann ein Schlüssel definiert werden, der beispielsweise dazu geeignet ist, um alle Artikel einer Firma aufzulisten – eine Schlüsseldefinition, die also nur aus dem Feld „Firma“ besteht?

Die Frage ist leicht beantwortet: Indem schlicht eine weitere Schlüsselliste im RPG definiert wird. Denn ein RPG-Programm unterstützt eine beliebige Anzahl von Schlüssellistendefinitionen – auch für eine Datei!

Schauen Sie sich dazu den folgenden Code-Ausschnitt an. Hier wurde eine weitere Schlüsselliste für die Datei DemoArt definiert. Diese besteht allein aus dem Feld x1firm.

```
     *------------------------------------------------------------------------*
     * Satz + Schlüssel Definitionen                                          *
     *------------------------------------------------------------------------*

c      DEMOK          KLIST
c                     kfld                    x1firm
c                     kfld                    x1teil

c      DEMOK2         KLIST
c                     kfld                    x1firm
```

Mehrere Schlüssellisten

3.2.6
Seite 12

Nachdem nun einiges über die Schlüsseldefinition geschildert wurde, muss noch erwähnt werden, dass der Pointer in der Datei positioniert werden muss. Das bedeutet, dass mit Hilfe der Schlüsseldefinition in der Datei dort aufgesetzt wird, wo die Daten eingelesen werden sollen. Das Positionieren kann in RPG mittels der beiden Positionierangaben setLL (Set Lower Limit) und setGT (Set Greater Limit) erfolgen.

Im folgenden Beispiel wurde setLL verwendet. Der Zeiger innerhalb der Datei wurde auf die Stelle gesetzt, die als Erste der Angabe der Schlüsselfelder entspricht.

Mit dem Vorbelegen des Schlüsselfeldes x1Firm = ‚01' wird angegeben, dass alle Datensätze in der Datei verarbeitet werden sollen, deren Firmennummer dem Wert des Schlüsselfeldes entspricht – also nur die Artikel der Firma 01. Wie bereits erwähnt, müssen Entwickler dafür Sorge tragen, dass alle Schlüsselfelder in der Schlüsselliste bedient werden – also gefüllt sind. Wenn zum Beispiel alle Artikel einer Firma aufgelistet werden sollen, dann muss dazu eine Schlüsseldefinition verwendet werden, die alleine aus der Firmennummer besteht. In diesem Beispiel ist das die Schlüsselliste DemoK2.

RPGREAD.RPGLE | DEMOART.PF | RPGREADE.RPGLE

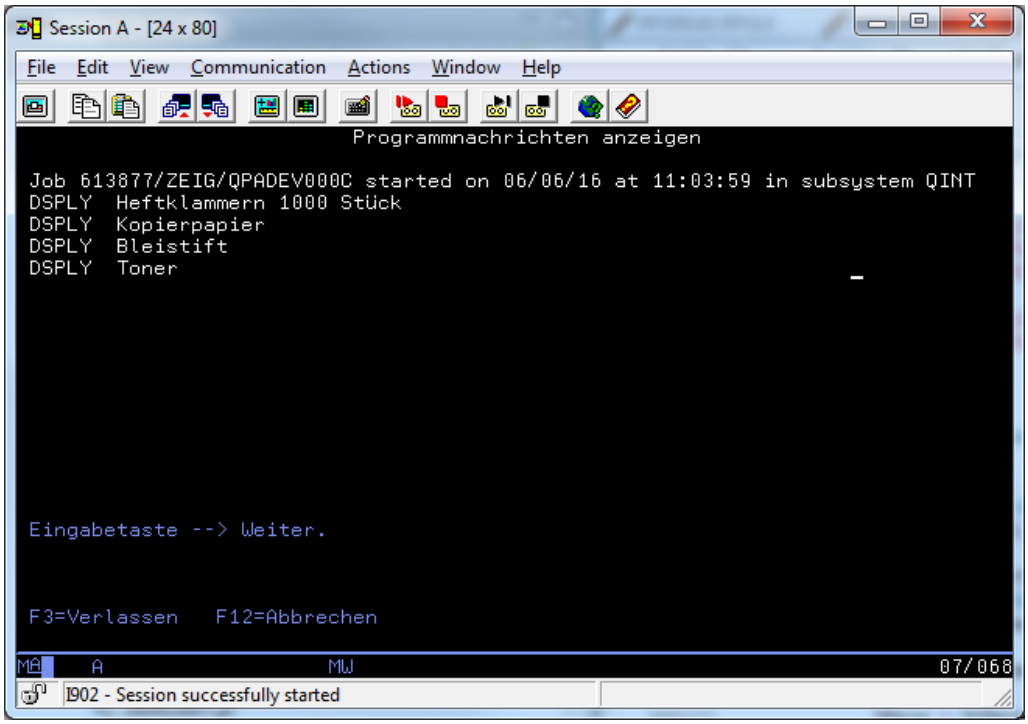
Zeile 37 Spalte 42 Einfügen

```
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8.
000112
000113      *---------------------------------------------------------------------*
000114      * Satz + Schlüssel Definitionen                                       *
000115      *---------------------------------------------------------------------*
000116
000117     c     DEMOK          KLIST
000118     c                    kfld                    x1firm
000119     c                    kfld                    x1teil
000120
000121     c     DEMOK2         KLIST
000122     c                    kfld                    x1firm
000124
000131      /FREE
000151
000152       // Schlüsselfelder Werte zuweisen
000153       x1Firm = '01';
000155
000156       // Positionieren in der Datei
000157       setll DemoK2 DemoArt;
000158
000159         If Not %Eof(DemoArt);
000160         // Einlesen des ersten Datensatzes
000161           ReadE DemoK2 DemoArt;
000162         // Leseschleife
000163            DoW Not %Eof(DemoArt);
000164
000165              DBeze = X1Beze;
000166              dsply DBeze;
000167         // Einlesen der weiteren Datensätze
000168              ReadE DemoK2 DemoArt;
000169            Enddo;
000170       Endif;
```

setLL-Beispiel

Dieser Beispiel-Code verarbeitet alle Datensätze in der Datei DemoArt, deren Firma den Wert „01“ hat.

Das Ergebnis: Alle Datensätze der Firma 01 werden aufgelistet.



Ergebnisanzeige

In diesem Fall hätte auch eine Abwandlung der Schlüsseldefinition genutzt werden können.

Im vorherigen Beispiel gibt es nur ein Schlüsselfeld, was die Verwendung der Schlüsselliste überflüssig macht. Denselben Effekt im Auslesen aller Datensätze für die Firma 01 liefert auch das folgende Beispiel. Hier wird ein Feld als Schlüssel genutzt, das sowohl den Namen des Feldes aus der Datei als auch den Namen einer beliebigen Variablen oder Konstanten tragen kann. Hier ist wieder wichtig: Die Feldart muss dem Feld in der Datei entsprechen!

3.2.6
Seite 14

RPGREAD.RPGLE | DEMOART.PF | RPGREADE.RPGLE

```
Zeile 26      Spalte 48    Einfügen
      ....+....1....+....2....+....3....+....4....+....5....+....6....
000121     c      DEMOK2         KLIST
000122     c                     kfld                    x1firm
000124
000131      /FREE
000151
000152       // Schlüsselfelder Werte zuweisen
000153       x1Firm = '01';
000155
000156       // Positionieren in der Datei
000157       setll X1Firm DemoArt;
000158
000159         If Not %Eof(DemoArt);
000160         // Einlesen des ersten Datensatzes
000161           ReadE X1Firm DemoArt;
000162         // Leseschleife
000163            DoW Not %Eof(DemoArt);
000164
000165              DBeze = X1Beze;
000166              dsply DBeze;
000167         // Einlesen der weiteren Datensätze
000168              ReadE X1Firm DemoArt;
000169            Enddo;
000170       Endif;
```

Positionieren und Lesen mit Feldschlüssel

Wenn dieses Programm aufgerufen wird, führt das zum selben Ergebnis wie bei der Nutzung mit der Schlüsselliste, die nur die Firmennummer beinhaltet.

An anderer Stelle werden die verschiedenen Möglichkeiten in der Schlüsselverwendung behandelt.

Zusammenfassend kann Folgendes festgehalten werden: Es können mit der Kombination einer Positionierungsanweisung (setLL oder setGT) und der Leseanweisung READE gezielt Einzelsätze oder alle Datensätze aus einer Datei verarbeitet werden, die der Schlüsseldefinition entsprechen.

3.2.6.1.3 CHAIN – Positionieren und Lesen eines Datensatzes

Im vorherigen Beispiel wurde gezeigt, wie die Kombination aus einem Positionierungsbefehl und einer Leseanweisung für den zielgerichteten Zugriff auf Datensätze ermöglicht wird, die dem vorgegebenen Schlüsselbegriff entsprechen.

IBM bietet eine weitere RPG-Anweisung, um auf den Dateiinhalt abzielen und zum Beispiel mit der Firma und der Artikelnummer auf einen Artikelstammsatz zugreifen zu können – CHAIN.

Mit der Anweisung CHAIN wurden quasi zwei Fliegen mit einer Klappe geschlagen:

1. Das Positionieren innerhalb der Datei mit dem vorgegebenen Schlüssel
2. Das Lesen des betreffenden Datensatzes, der den Schlüsselwerten entspricht

Bei der Verwendung von CHAIN müssen folgende Angaben getätigt werden:

1. Angabe des Schlüssels (Schlüsselliste, Datenstruktur oder Einzelwert)
2. Angabe der Datei, die gelesen werden soll
3. Optional kann eine Bezugszahl verwendet werden, mit der festgestellt werden kann, ob die CHAIN-Ausführung erfolgreich war (Hinweis: Für den Fall bietet sich die Verwendung einer BIF anstelle der Bezugszahlen an. Insbesondere in älteren RPG-Varianten ist die Nutzung der Bezugszahl als „Erfolgskennzeichen“ weit verbreitet. Moderner RPG-Stil setzt auf die Vermeidung der Bezugszahl und die Nutzung der Built-In-Functions. Dazu später mehr.)

Nun soll CHAIN in Aktion betrachtet werden.

3.2.6
Seite 16

Im Folgebeispiel wird erneut die Datei DemoArt genutzt und mit der Firma und der Artikelnummer (5000) direkt auf den gewünschten Datensatz zugegriffen. Die Artikelbezeichnung wird dazu angezeigt.

RPGCHAIN.RPGLE

```
Zeile 36      Spalte 54    Einfügen
      ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8.
000104     FDemoArt   IF   E           K DISK
000105
000106     dDBeze            S             50    inz
000112
000113      *---------------------------------------------------------------------*
000114      * Satz + Schlüssel Definitionen                                       *
000115      *---------------------------------------------------------------------*
000116
000117     c      DEMOK         KLIST
000118     c                    kfld                    x1firm
000119     c                    kfld                    x1teil
000120
000121     c      DEMOK2        KLIST
000122     c                    kfld                    x1firm
000124
000131      /FREE
000151
000152       // Schlüsselfelder Werte zuweisen
000153       x1Firm = '01';
000154       x1Teil = '5000';
000155
000156       // Positionieren und Lesen der Datei mit CHAIN
000157       chain DemoK2 DemoArt;
000158
000159       If  %Found(DemoArt);
000165            DBeze = X1Beze;
000166            dsply DBeze;
000170       Endif;
```

CHAIN-Beispiel

Das Ergebnis ist die Anzeige des Artikeltextes „Heftklammer“ – dem Text zu unserer vorgegebenen Artikelnummer 5000.

Im Vergleich zu setLL/READE besteht hier zwar weniger Codier-Aufwand, doch kann mit CHAIN nur ein Datensatz verarbeitet werden. Und zwar deshalb, weil CHAIN jedes Mal auf den gewünschten Satz positioniert, um diesen anschließend direkt einzulesen.

3.2.6.2 Schlüsseldefinitionen in RPG

Die Schlüssel und deren Bedeutung haben Sie in den Lesebeispielen bereits kennengelernt.

IBM bietet den Vorteil, dass ältere Techniken noch heute auf modernen Maschinen weitestgehend unterstützt werden. Was die einen jedoch als Vorteil betrachten, ist für andere ein Nachteil. Doch mit dieser Tatsache soll sich hier nicht weiter auseinandergesetzt werden.

Die Nutzung der Schlüssellisten kann als „klassisch" bezeichnet werden. Allein die Tatsache, dass eine Umsetzung der KLIST-/kfld-Anweisungen nicht in freies Format möglich ist, zeigt, wie IBM diese potenzielle Option der Schlüsseldefinitionen sieht – sie hat ausgedient und sollte im modernen RPG-Programm-Code vermieden werden.

Der Vollständigkeit halber sei sie hier jedoch erwähnt – denn in unzähligen Programm-Codes sind diese Schlüsseldefinitionen zu finden, und es muss damit umzugehen gewusst sein.

Ergänzung 39./2016

3.2.6
Seite 18

3.2.6.2.1 KLIST/kfld

Im Fix-Format-RPG steht die Anweisung KLIST zur Verfügung. Diese Anweisung bedarf stets einer zweiten Anweisung kfld, die zwingend nach KLIST angegeben werden muss.

KLIST teilt dem RPG-Programm mit, dass eine Schlüsselliste definiert werden soll. Dieser muss ein eindeutiger Name zugeordnet werden, hier: DemoK.

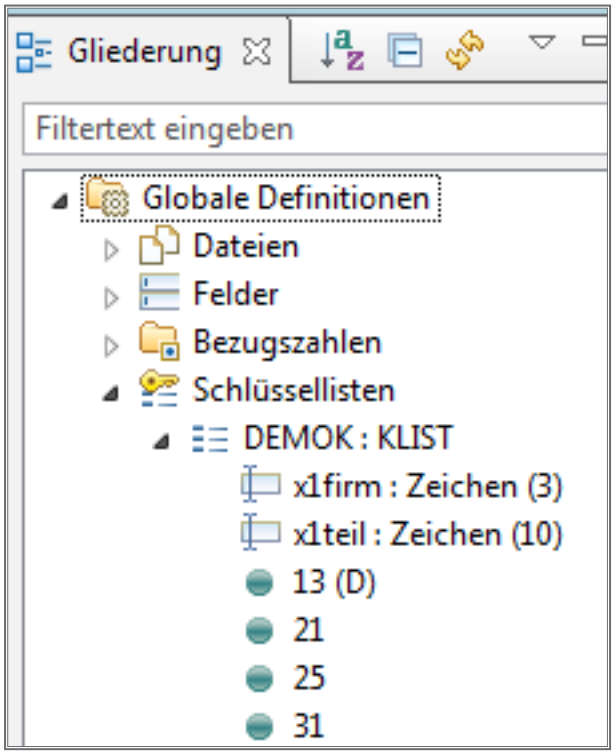
Diese Schlüsselliste wird mit den beiden Schlüsselfeldern definiert, die in der Datei existieren. Es werden dieselben Namen verwendet, wie sie auch in der Datei hinterlegt sind. Doch ist dies kein Muss, denn es können hier natürlich auch andere Feldnamen genutzt werden. Wichtig ist nur, dass diese mit der Definition in der Datei in Bezug auf die Feldart und die Feldlänge übereinstimmen.

RPGREAD.RPGLE | DEMOART.PF | RPGREADE.RPGLE

```
Zeile 35      Spalte 18    Einfügen
      ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8.
000103     H COPYRIGHT('© H.J. Zeig ZeBIS 2016')
000104     FDemoArt   IF   E           K DISK
000105
000106     dDBeze            S             50    inz
000112
000113      *---------------------------------------------------------------------*
000114      * Satz + Schlüssel Definitionen                                       *
000115      *---------------------------------------------------------------------*
000116
000117     c     DEMOK         KLIST
000118     c                   kfld                    x1firm
000119     c                   kfld                    x1teil
000120
000131      /FREE
000132
000133        // Positionieren in der Datei
000134       x1Firm = '01';
000135       setll DemoK DemoArt;
000136
000137         If Not %Eof(DemoArt);
000138         // Einlesen des ersten Datensatzes
000139           ReadE DemoK DemoArt;
000140         // Leseschleife
000141            DoW Not %Eof(DemoArt);
000142
000143              DBeze = X1Beze;
000144         // Einlesen der weiteren Datensätze
000145              ReadE DemoK DemoArt;
000146            Enddo;
000147       Endif;
000148
000149       *inlr = *on;
000150
```

Schlüssellistendefinition mit KLIST

RDI zeigt in der Gliederungssicht weitere Details zur Schlüsselliste an. Hier können zum Beispiel die in der Liste enthaltenen Schlüsselfelder und deren Definitionen angezeigt werden.



Schlüssellistenanzeige in der RDI-Gliederungssicht

Nun wurden die Datei und die Schlüsselliste definiert. Es fehlt nur noch das Zuweisen eines Wertes in die gewünschten Schlüsselfelder.

Dies erfolgt hier, indem den Schlüsselfeldern die folgenden Werte zugewiesen werden:

x1Firm = ‚01‘

x1Teil = ‚5000‘

```
000133        // Schlüsselfelder Werte zuweisen
000134        x1Firm = '01';
000135        x1Teil = '5000';
```

Wertzuweisung im Schlüsselfeld

3.2.6
Seite 20

Achtung!

Hier ist eine Besonderheit bei der Verwendung von KLIST zu beachten. Zusammen mit KLIST wurde dem RPG-Programm im Beispiel mitgeteilt, dass die Schlüsselliste aus zwei Feldern besteht, die zwingend zusammen genutzt werden müssen. Das bedeutet auch, dass beide Felder für das korrekte Verarbeiten mit einem Wert belegt werden müssen. Würde zum Beispiel nur einem der Schlüsselfelder (x1firm) ein Wert zugewiesen, dann führe dies dazu, dass die Schlüsseldefinition nicht vollständig ist und damit die Verarbeitung nicht den gewünschten Effekt erzielt.

Das kann durchaus als Nachteil bei der Verwendung von KLIST angesehen werden.

Schlüssellisten können mehrfach für eine Datei in RPG angegeben werden, womit zum Beispiel alle möglichen Vorkommen abgebildet werden können.

Schauen Sie sich dazu den folgenden Code-Ausschnitt an. Hier wurde eine weitere Schlüsselliste für die Datei DemoArt definiert. Diese besteht allein aus dem Feld x1firm.

```
     *-------------------------------------------------------------------------*
     * Satz + Schlüssel Definitionen                                           *
     *-------------------------------------------------------------------------*

     c     DEMOK         KLIST
     c                   kfld                    x1firm
     c                   kfld                    x1teil

     c     DEMOK2        KLIST
     c                   kfld                    x1firm
```

Mehrere Schlüssellisten

Theoretisch könnte beim Positionieren und Lesen in einer Schleife auch eine Kombination aus verschiedenen Schlüssellisten verwendet werden. Anhand der beiden im Beispiel gezeigten Schlüssellisten könnte die Schlüsselliste DemoK dazu genutzt werden, um in einer Artikelstammdatei auf den Satz zu positionieren, der der Firma/der Teilenummer entspricht. Ab dieser könnten dann alle weiteren Sätze verarbeitet werden, die in der Firma angelegt sind.

Nachfolgendes Code-Beispiel zeigt dies.

RPGREADE2.RPGLE

```
 Zeile 42      Spalte 57    Einfügen
      ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+..
000115      *----------------------------------------------------------------------
000116
000117     c     DEMOK         KLIST
000118     c                   kfld                    x1firm
000119     c                   kfld                    x1teil
000120
000121     c     DEMOK2        KLIST
000122     c                   kfld                    x1firm
000124
000131      /FREE
000151
000152       // Schlüsselfelder Werte zuweisen
000153       x1Firm = '01';
000154       x1Teil = '5000';
000155
000156       // Positionieren in der Datei
000157       setll DemoK  DemoArt;
000158
000159         If Not %Eof(DemoArt);
000160         // Einlesen des ersten Datensatzes
000161           ReadE DemoK2 DemoArt;
000162         // Leseschleife
000163            DoW Not %Eof(DemoArt);
000164
000165              DBeze = X1Beze;
000166              dsply DBeze;
000167         // Einlesen der weiteren Datensätze
000168              ReadE DemoK2 DemoArt;
000169            Enddo;
000170       Endif;
```

Positionieren und Lesen mit verschiedenen Schlüssellisten

3.2.6.2.2 Schlüsseldatenstrukturen

Moderner RPG-Programm-Code bedient sich bei der Schlüsseldefinition der Verwendung von Datenstrukturen. Diese Datenstrukturen bieten eine hohe Flexibilität – können sie doch universell eingesetzt werden und mit wenigen Erweiterungen flexibel die Anzahl der Schlüsselfelder bzw. deren Verwendung steuern.

Datenstrukturen sind in RPG ein wichtiger Baustein und können neben den Schlüsseln auch in anderen Bereichen zum Einsatz kommen.

Sehen Sie hier ein Beispiel für eine erste Schlüsseldefinition auf der Basis einer Datenstruktur.

Folgende Angaben müssen getätigt werden:

- Name der Datenstruktur vergeben
- Kennzeichnen, dass es sich bei der Definition um eine Datenstruktur handelt
- Bezug festlegen – Datei angeben
- Verwendungsart (*KEY) hinterlegen
- Optional kann eine qualifizierte Nutzung definiert werden (QUALIFIED).

Im folgenden Code-Ausschnitt wurde eine Datenstruktur Demo_K definiert.

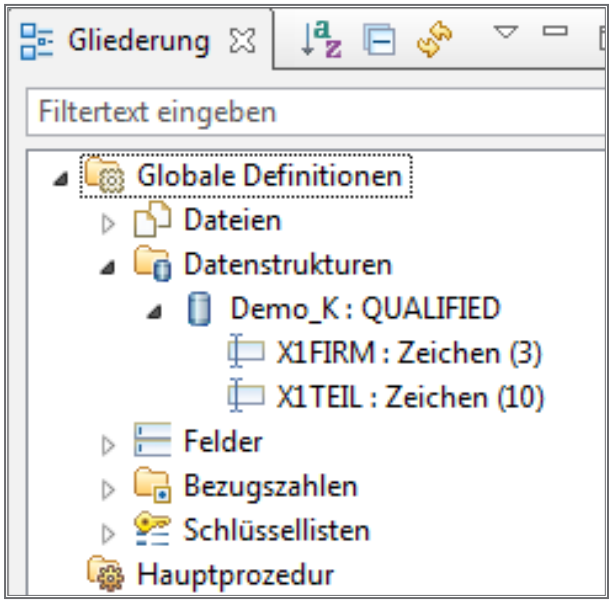
Tipp:

Verwenden Sie nach Möglichkeit einen Namen, der auf die Datei und die Verarbeitungsart schließen lässt. Ich nutze zum Beispiel für Schlüsseldateien gerne die Endung „_K“. Damit kann ich im Programm-Code direkt und schnell die Bedeutung ableiten.

```
RPGREADE2.RPGLE   WW300R.RPGLE   RPGCHAIN2.RPGLE
Zeile 32    Spalte 74    Einfügen
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8...
000103     H COPYRIGHT('© H.J. Zeig ZeBIS 2016')
000104     FDemoArt   IF   E           K DISK
000105
000106     dDBeze            S             50    inz
000107
000108      // Schlüsseldatenstruktur
000109     d Demo_K        e ds                  extName(DEMOArt:*Key)
000110     d                                     qualified
000112
000113      *---------------------------------------------------------------------*
000114      * Satz + Schlüssel Definitionen                                       *
000115      *---------------------------------------------------------------------*
000116
000117     c      DEMOK         KLIST
000118     c                    kfld                    x1firm
000119     c                    kfld                    x1teil
000120
000121     c      DEMOK2        KLIST
000122     c                    kfld                    x1firm
```

Schlüsseldatenstruktur

Bei Ansicht der Datenstruktur in der Gliederung lassen sich die in der Datenstruktur enthaltenen Schlüsselfelder und deren Feldart erkennen.



Datenstrukturansicht in der Gliederung

Der wesentliche Unterschied zur Definition mit KLIST liegt darin, dass nirgends ein Feldname angegeben wurde. Mit der *KEY-Definition ruft RPG die Dateibeschreibung ab und ermittelt so automatisch die Schlüsselfelder, die in der angegebenen Datei enthalten sind. Diese Schlüsselinformationen werden in die Schlüsseldatenstruktur übertragen und können nun flexibel genutzt werden.

3.2.6
Seite 24

Die Zuweisung der Werte in die zu nutzenden Schlüsselfelder muss auch bei der Nutzung der Schlüsseldatenstrukturen erfolgen. Diese haben den Vorteil, dass im Zusammenspiel mit Built-In-Functions die Schlüsseldatenstrukturen effektiv eingesetzt werden können.

Dazu ein Beispiel:

RPGCHAIN2.RPGLE

```
Zeile 30      Spalte 61    Einfügen
      ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8.
000103     H COPYRIGHT('© H.J. Zeig ZeBIS 2016')
000104     FDemoArt   IF   E           K DISK
000105
000106     dDBeze            S             50    inz
000107
000108      // Schlüsseldatenstruktur
000109     d Demo_K        e ds                  extName(DEMOArt:*Key)
000112
000113      *---------------------------------------------------------------------*
000114      * Satz + Schlüssel Definitionen                                       *
000115      *---------------------------------------------------------------------*
000116
000117     c      DEMOK         KLIST
000118     c                    kfld                    x1firm
000119     c                    kfld                    x1teil
000120
000121     c      DEMOK2        KLIST
000122     c                    kfld                    x1firm
000124
000131      /FREE
000151
000152       // Schlüsselfelder Werte zuweisen
000153       x1Firm = '01';
000154       x1Teil = '5000';
000155
000156       // Positionieren und Lesen der Datei mit CHAIN
000157       chain %kds(Demo_K:2) DemoArt;
000158
000159       If  %Found(DemoArt);
000165            DBeze = X1Beze;
000166            dsply DBeze;
000170       Endif;
```

CHAIN mit Schlüsseldatenstruktur und %KDS

In obigem Beispiel wurde die BIF %KDS verwendet, die es erlaubt, zusammen mit der Schlüsseldatenstruktur auf den Dateiinhalt DemoArt zuzugreifen. Die BIF %KDS verfügt über die folgenden Parameter:

- Name der Schlüsseldatenstruktur (im Beispiel Demo_K)
- Anzahl der Schlüsselfelder (im Beispiel handelt es sich um 2 Schlüsselfelder– dabei ist die Reihenfolge der Schlüsselfelder aus der Datei maßgeblich!)

Wenn nur mit einem Schüsselfeld (der Firma) gearbeitet werden soll, dann genügt ein einfaches Austauschen der „2“ (für zwei Schlüsselfelder) durch eine „1“. Eine Mehrfachdefinition von Schlüsseln, wie bei der Nutzung von KLIST, gibt es hier nicht bzw. ist nicht erforderlich!

3.2.7 Datenmanipulation mit RPG

Im vorhergehenden Abschnitt wurde das Lesen der Daten behandelt, doch weil auch die Datenmanipulation ein wichtiger Schritt in einer Anwendung ist, soll diese Thematik im Folgenden erläutert werden.

Datenmanipulation gliedert sich in

- das Schreiben eines neuen Datensatzes,
- das Verändern eines bestehenden Datensatzes und
- das Löschen eines Datensatzes.

Für diese Formen der Datenmanipulation stehen entsprechende RPG-Anweisungen zur Verfügung. Neben den typischen RPG-nativen Anweisungen zum Schreiben, Verändern und Löschen von Daten können im RPG auch SQL-Derivate für Datenmanipulationen verwendet werden. Hier soll sich aber zunächst auf den nativen RPG-Ansatz beschränkt und nur ein erster kleiner Einblick in die SQL-Derivate gegeben werden. Details zu embedded SQL sind in einem separaten Kapitel zu finden.

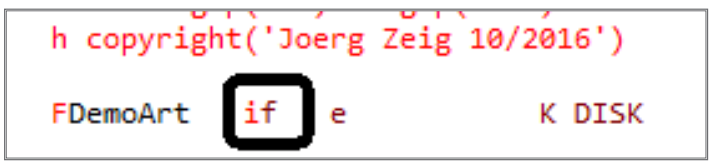
Vor der Betrachtung einzelner Fälle der Datenmanipulation müssen zunächst die Voraussetzungen geschaffen werden, um in einem RPG-Programm die Dateidaten verändern zu können.

Wie bereits beschrieben, müssen in den Dateibestimmungen in einem RPG-Programm alle Dateien, die im Programm verwendet werden sollen, definiert werden. Dazu ist die Verwendungsart anzugeben, und zwar in der Form, ob die Daten gelesen oder geändert oder ob neue Datenelemente in eine Datei eingefügt werden sollen.

Die Dateibestimmungen legen also generell die Verwendungsart und die Verarbeitungsformen fest.

Folglich müssen die notwendigen Einstellungen auf der Ebene der Dateibestimmungen, mit denen eine Datenmanipulation möglich wird, betrachtet werden.

Folgende Abbildung zeigt eine Beispieldatei, die in einem RPG-Programm nur lesend eröffnet wird. Dies ist am Eintrag „i“ (Input) an Stelle 17 (Dateiart) im Fix-Format-RPG erkennbar.



3.2.7
Seite 2

Mit dieser Eintragung kann im RPG-Programm auf den Inhalt der Datei DemoArt nur lesend zugegriffen werden. Eine Veränderung der Dateiinhalte oder ein Hinzufügen neuer Datensätze ist mit dieser Einstellung nicht möglich.

Wenn Sie RPG im Free-Format nutzen, dann gelten dieselben Kriterien wie für die Dateibestimmungen im Fix-Format.

Das entsprechende Beispiel für ein Free-Format-RPG finden Sie hier:

```
//
ctl-opt copyright('Joerg Zeig 10/2016') ;

// Dateibestimmung
dcl-f DemoArt Disk usage(*input) keyed;

*inlr = *on;
```

Was passiert, wenn eine Datei nicht korrekt in Bezug auf die Verwendungsart definiert wird?

Ein kleines Beispiel anhand eines RPG-Programms, in dem die Datei DemoArt definiert wurde, veranschaulicht dies.

Wird nun versucht, im Programm eine Datensatzveränderung vorzunehmen, dann endet das bereits zum Zeitpunkt der Programmerstellung mit einer fehlerhaften Compilierung. Und zwar deshalb, weil der Compiler prüft, ob alle Voraussetzungen erfüllt sind, die eine Datenmanipulation zulassen.

Folgende Abbildung zeigt einen typischen Wandlungsfehler, der dann ausgegeben wird, wenn versucht wird, eine zum Input definierte Datei mit einer Änderungsanweisung fortzuschreiben.

ID	Nachricht
RNS9308	Umwandlung gestoppt. Fehler mit Bewertungsstufe 30 im Programm gefunden.
RNF5198	Datei in Faktor 2 ist für eine Operation UPDATE oder DELETE nicht zulässig.

Der Compilierfehler RNF5198 zeigt an, dass die Daten in der Datei nicht verändert werden dürfen.

Um das Verändern bestehender Dateninhalte zu ermöglichen, muss in der Dateibestimmung in Spalte 17 (Dateiart) der Eintrag „U“ (Update) getätigt werden.

Folglich sollte die Dateibestimmung so ausschauen:

```
FArtikRf   uF   e           K DISK
```

Hier ist das korrespondierende Beispiel Free-Format zu sehen:

```
// Dateibestimmung
dcl-f ArtikRf Disk usage(*update) keyed;
```

Mit diesen Dateibestimmungseinstellungen wird immenser Einfluss auf einen der performanceintensivsten Prozesse bei der Ausführung eines Programms ausgeübt: das Öffnen einer Datei und das Herstellen des erforderlichen Zugriffspfads auf den Inhalt derselben. Das reine Öffnen wird dabei differenziert nach einem Öffnen zum Lesen und einem Öffnen zum Fortschreiben. Der Zugriffspfad wird dem RPG-Programm als Grundlage für das Arbeiten mit den Dateiinhalten je Programmaufruf zur Verfügung gestellt. Hier sollten Sie bereits beim Programmdesign aufmerksam sein, denn eine geschickte Behandlung der Dateien und der Zugriffspfade in einem Programmablauf kann sich entscheidend auf die Anwendungs-Performance auswirken. Als Stichworte seien hier gemeinsam genutzte Zugriffspfade und wiederverwendbare Zugriffspfade genannt. Dazu später mehr.

Wichtig: Für das Verändern bestehender Dateiinhalte in den Dateibestimmungen muss die Datei mit der Dateiart „Update“ definiert werden.

Damit sind folgende Formen der Datenmanipulation möglich:

- Ändern bestehender Datensätze (zum Beispiel das Anpassen des Artikeltextes) und
- Löschen bestehender Datensätze (hier kann wahlweise ein Feld oder eine Spalte initialisiert oder geleert oder der gesamte Datensatz gelöscht werden).

Mit dieser Einstellung ist es nicht möglich, neue Datensätze hinzuzufügen. Hierzu ist eine weitere spezielle Definition erforderlich!

Um neue Datensätze in einer Datei hinzufügen zu können, muss in Spalte 20 (Hinzufügen neuer Sätze) ein „A“ eingetragen werden. Im Free-Format-RPG ist hier die Verwendung *Output anzugeben.

Nachfolgend finden Sie je ein Beispiel für eine Dateibestimmung, mit der neue Datensätze in eine Datei mittels RPG geschrieben werden können.

Fix-Format-Definition für das Hinzufügen neuer Datensätze:

```
FArtikRf   uF A e           K DISK
```

Free-Format-Definition für das Hinzufügen neuer Datensätze:

```
// Dateibestimmung
dcl-f ArtikRf Disk usage(*output) keyed;
```

Sobald die Grundlagen für die Datenmanipulation in den Dateibestimmungen geschaffen wurden, ist die Beschäftigung mit den RPG-Anweisungen möglich.

3.2.7
Seite 4

3.2.7.1 Hinzufügen von neuen Datensätzen mit RPG nativ

Mit RPG können neue Datensätze in eine Datei/Tabelle eingefügt werden, wobei es keine Rolle spielt, ob zum Hinzufügen eine physische Datei, eine SQL-Tabelle, eine logische Datei oder das entsprechende SQL-Pendant verwendet wird.

Beim Hinzufügen muss allerdings dafür gesorgt werden, dass alle notwendigen Informationen eines Datensatzes korrekt belegt werden. Das Schreiben eines neuen Datensatzes mit RPG erfolgt auf der Basis des gesamten Datensatzes auf Satzformatebene.

Mit der Dateibestimmung und der dort anzugebenden Dateidefinition als Ausgabedatei bzw. dem Aktivieren des Schalters „Neue Datensätze“ wurden die Grundlagen für neue Datensätze geschaffen.

Die Anweisung, mit der neue Datensätze mit RPG geschrieben werden können, heißt Write.

Write dient allgemein zum Schreiben neuer Sätze in einer Datei, wobei die Dateiart erst einmal keine Rolle spielt. So kann zum Beispiel mit Write ein Datensatz in eine physische Datei oder eine logische Datei geschrieben werden sowie ein Datensatz in eine Display File geschrieben oder ein Datensatz in einer Druckerdatei ausgegeben werden. In diesem Abschnitt soll sich auf das Schreiben von Datensätzen in Datenbankdateien (physische Datei, logische Datei, SQL-Tabelle) beschränkt werden.

Mit der RPG-Anweisung Write (neuen Datensatz schreiben) wird wahlweise der Dateiname oder das Satzformat angegeben.

Zudem gibt es mehrere Optionen, wie die einzelnen Felder oder Spalten des Datensatzes gefüllt werden können.

In älteren RPG-Programmen ist es weit verbreitet, dass die Dateifelder direkt angesprochen werden, dass diese in einer Anweisung dediziert gefüllt werden und dass anschließend der Write-Prozess basierend auf den zuvor gefüllten Feldern vorgenommen wird.

Dies soll ein einfaches Beispiel veranschaulichen.

Die Datei hat folgenden Aufbau:

Datei ZEIG/DEMOART (6 Felder)

Name	Satz	Typ	Länge	Text
DAFIRM	RDEMOART	Zeichen	2	Firma
DATEIL	RDEMOART	Zeichen	10	Teilenummer
DABEZE	RDEMOART	Zeichen	50	Bezeichnung
DALAGO	RDEMOART	Zeichen	3	Lagerort
DABEST	RDEMOART	Gepackt dezimal	9,0	Bestand
DAAKTI	RDEMOART	Zeichen	1	Kz. Aktiv

Es folgt ein Programmbeispiel, mit dem ein neuer Datensatz (Artikelstammsatz) in die Datei DemoArt geschrieben werden kann.

Die Datei wurde so definiert, dass neue Sätze hinzugefügt werden können. Des Weiteren wird dort jedes Dateifeld direkt gefüllt. Zuvor muss aber mit der Anweisung Clear sichergestellt werden, dass alle möglichen Feldinhalte, die ggf. durch eine vorhergehende Verarbeitung geladen wurden, geleert werden.

```
h copyright('Joerg Zeig 10/2016')

FDemoArt   uF A e           K DISK

d Rpg0001         pr
d p_Firm                        2

d Rpg0001         pi
d p_Firm                        2

 /Free

  // Satz bereinigen
  clear rDemoArt;
```

Anschließend wird jedem einzelnen Feld der gewünschte Zielwert zugewiesen. Dabei spielt es keine Rolle, ob der Wert mit Hilfe einer Variablen gesetzt (zum Beispiel das Füllen der als Parameter übergebenen Firmennummer) oder fix codiert wird.

```
DaFirm = p_Firm;    // Zuweisung über Parameter
DaTeil = '5000';
DaBeze = 'USB Scanner';
DaLago = *blank;
DaBest = *zero;
DaAkti = 'A';
```

Nach dem Zuweisen der Feldwerte wird der gesamte Datensatz geschrieben. Dazu wird die Anweisung Write, gefolgt vom Datei- oder Satzformatnamen, verwendet. Meist werden Sie hier den Satzformatnamen verwenden. Dieser ist bei extern beschriebenen Dateien einzusetzen. Sollten Sie im RPG-Programm programmbeschriebene Dateien verarbeiten, dann müssen Sie zusammen mit Write den Dateinamen angeben. Allerdings lautet die IBM-Empfehlung, bei neuen Programmen und neuen Datenbankelementen (Dateien) keine programmbeschriebenen Dateien zu verwenden.

Deshalb soll sich hier ausschließlich auf extern beschriebene Dateien beschränkt werden.

3.2.7
Seite 6

Der gesamte Block sieht folgendermaßen aus:

```
// Satz bereinigen
clear rDemoArt;

// Feldwerte zuweisen
DaFirm = p_Firm;   // Zuweisung über Parameter
DaTeil = '5000';
DaBeze = 'USB Scanner';
DaLago = *blank;
DaBest = *zero;
DaAkti = 'A';

// Neuen Datensatz schreiben
write rDemoArt;
```

Wenn Sie das Programmbeispiel nun aufrufen, dann finden Sie den neuen Datensatz in der Datei.

	DATEIL	DABEZE	DALAGO	DABEST	D
01	4000	Heftstreifen ...	001	114	A
01	4010	Locher ...	001	10	A
01	4020	Kopierer ...	002	5	A
01	4045	Atlas 2015 ...	01	2	P
01	5000	USB Scanner		0	A

Dies ist ein erster Erfolg beim Schreiben eines neuen Datensatzes mittels RPG.

Der Vollständigkeit halber wird hier der komplette Code gezeigt.

```
//*************************************************************
//-------------------------------------------------------------

// S T E U E R - B E S T I M M U N G E N
//-------------------------------------------------------------
h datfmt(*EUR)
h datedit(*DMY.)
h timfmt(*HMS:)
h decedit('0,')
h langid('DE')
h option(*SRCSTMT:*NoDebugIO)
h dftactgrp(*NO) actgrp(*NEW)
h copyright('Joerg Zeig 10/2016')

FDemoArt uF A e K DISK

d Rpg0001 pr
d p_Firm 2

d Rpg0001 pi
d p_Firm 2

  /Free

   // Satz bereinigen
   clear rDemoArt;

   // Feldwerte zuweisen
   DaFirm = p_Firm; // Zuweisung über Parameter
   DaTeil = '5000';
   DaBeze = 'USB Scanner';
   DaLago = *blank;
   DaBest = *zero;
   DaAkti = 'A';

   // Neuen Datensatz schreiben
   write rDemoArt;

   *InLR = *On;
   /end-free
```

Natürlich handelt es sich hier nur um ein einfaches Beispiel für den Schreibprozess in einem RPG-Programm.

Beim Schreiben eines neuen Datensatzes müssen folgende Punkte beachtet werden:

1. Stellen Sie sicher, dass alle Feldinhalte korrekt gefüllt werden. Werden beispielsweise Datumsfelder richtig bedient? Sind die Feldlängen und Feldinhalte insbesondere bei Charakterfeldern richtig?

2. Stellen Sie beim Schreiben sicher, dass die Datenbankinhalte nach dem Schreiben auch passen. Achten Sie in diesem Zusammenhang insbesondere auf das korrekte Füllen und Behandeln von Schlüsselfeldern. (Wenn Sie zum Beispiel das erste Schreibprogramm mehrfach aufrufen, dann wird jedes Mal ein neuer Datensatz „USB Scanner“ mit der Artikelnummer 5000 in die Datei geschrieben. Das ist so sicher nicht gewollt. Deshalb müssen bei der Programmerstellung und der Programmlogik auch die Besonderheiten der Datenbank und der Datenbankelemente berücksichtigt werden. Aber das ist in jeder Programmiersprache so und unabhängig vom RPG!)

3. Allgemein muss sichergestellt sein, dass die Anforderungen der Datenbank (Schlüsselfelder, relative Satznummern etc.) eingehalten und beachtet werden.

4. Stellen Sie sicher, dass Sie die korrekten Feldinhalte schreiben. Im Beispiel werden für die Zuweisungen die Feldnamen aus der Datei verwendet. Wenn diese im Programmverlauf etwa durch Leseprozesse verändert werden, müssen Sie sicherstellen, dass das Schreiben mit den gewünschten und korrekten Werten erfolgt. Abhilfe könnte hier die Qualifizierung der Zuweisungen schaffen (siehe nachfolgenden Abschnitt).

5. Stellen Sie sicher, dass Felder korrekt initialisiert werden (beachten Sie hier unter anderem auch die Null-Behandlung).

Im Anschluss an das erste Programmbeispiel folgen hier einige Ergänzungen zum Schreiben neuer Datensätze.

RPG bietet die Möglichkeit, Datenstrukturen für alle möglichen Bereiche einzusetzen. Auch für das Schreiben von neuen Datensätzen können Datenstrukturen genutzt werden.

Wie das geht, soll anhand des nachfolgenden Beispiels aufgezeigt werden.

Der Vorteil bei der Verwendung von Datenstrukturen im Zusammenhang mit der Neuanlage von Datensätzen liegt in der klaren Abgrenzung innerhalb des RPG-Programms.

Und zwar deshalb, weil mit einer Datenstruktur die Behandlung der Daten innerhalb des Programms festgelegt werden kann. So können zum Beispiel auf der Ebene der Datenstruktur auf eine Datei referenziert, die Verwendung der Datenstruktur klar vorgegeben (zum Beispiel *Output) und die einzelnen Felder der Datenstruktur qualifiziert verwendet werden.

Die Definition einer Datenstruktur für Ausgabezwecke kann folgendermaßen aussehen:

```
d Arti_O E DS ExtName(DemoArt:*Output)
d Qualified
```

Hier wurde die Datenstruktur Arti_O definiert. Der Name ist frei zu vergeben. Bei der Datenstruktur wurde darauf verwiesen, dass die Basis für die Datenstruktur in einer externen Beschreibung zu finden ist (Angabe „E“ in Spalte 22). Mit dem Schlüsselwort ExtName wird sodann auf die externe Beschreibung verwiesen. Im Beispiel ist das die Datei DemoArt.

Da die Datenstruktur gezielt zum Schreiben von Daten verwendet werden soll, muss hier die Ergänzung *Output angegeben werden.

Anmerkung:

Datenstrukturen können im RPG auch zur reinen Eingabe (*Input) oder variabel als Schlüsseldatenstrukturen (*Key) genutzt werden.

Damit die Komponenten der Datenstruktur im RPG-Programm gezielt angesprochen und verarbeitet werden können, muss die Datenstruktur als „qualifiziert“ definiert werden. Damit müssen auch die einzelnen Bestandteile der Datenstruktur (im Beispiel sind das die einzelnen Felder der zugrunde liegenden Datei DemoArt) gezielt bzw. qualifiziert verwendet werden.

Basierend auf der Datei DemoArt bauen sich dann die Feldnamen der Datenstruktur wie folgt auf:

Name der Datenstruktur.Feldname

Als Beispiel:
Arti_O.DaFirm (Firma)

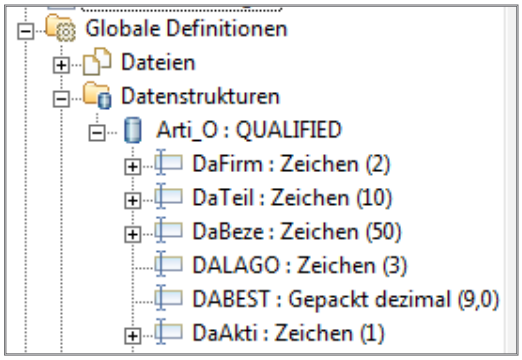
Arti_O.DaTeil (Teilenummer)

Etc.

3.2.7
Seite 10

Ein weiterer Vorteil bei der Verwendung von Datenstrukturen ist, dass sich die Datenstrukturen im RDI in der Gliederung wunderbar betrachten lassen.

Hier ist zum Beispiel der Definitionsbereich für die Ausgabedatenstruktur Arti_O zu sehen:



Basierend auf den Datenstrukturvorgaben werden die neuen qualifizierten Feldnamen auch im Programm genutzt. Folgender Codeabschnitt zeigt die Zuweisungen zu den einzelnen Feldern der Datenstruktur:

```
// Feldwerte zuweisen
Arti_O.DaFirm = p_Firm;    // Zuweisung über Parameter
Arti_O.DaTeil = p_Teil;
Arti_O.DaBeze = p_Beze;
Arti_O.DaAkti = 'A';
```

Anmerkung:

Um das erste RPG-Programmbeispiel erweitern zu können, werden die Parameterdefinitionen um die Felder „Artikelnummer“ und „Artikelbeschreibung“ ergänzt. Diese sind im vorhergehenden Beispiel als Parameterzuweisungen angepasst und werden in die Einzelfelder der Ausgabedatenstruktur Arti_O übertragen.

Das Schreiben des Datensatzes wird auch hier mit der Anweisung Write, gefolgt vom Satzformatnamen und dem Namen der Ausgabedatenstruktur Arti_O, durchgeführt.

```
// Neuen Datensatz schreiben
write rDemoArt Arti_O;
```

Der Vollständigkeit halber wird das gesamte Codebeispiel aufgezeigt.

```
FDemoArt   uF A e           K DISK

d Rpg0002         pr
d p_Firm                     2
d p_Teil                    10
d p_Beze                    50

d Rpg0002         pi
d p_Firm                     2
d p_Teil                    10
d p_Beze                    50

d Arti_O        E DS                  ExtName(DemoArt:*Output)
d                                     Qualified

 /Free

  // Satz bereinigen
  clear rDemoArt;    // auf Satzformatebene oder
  clear Arti_O;      // auf Datenstrukturebene

  // Feldwerte zuweisen
  Arti_O.DaFirm = p_Firm;    // Zuweisung über Parameter
  Arti_O.DaTeil = p_Teil;
  Arti_O.DaBeze = p_Beze;
  Arti_O.DaAkti = 'A';

  // Neuen Datensatz schreiben
  write rDemoArt Arti_O;

  *InLR = *On;
```

Ein weiterer Vorteil bei der Nutzung der Datenstrukturen ist, dass diese sehr elegant im Debug verwendet werden können.

IBM hat Entwickler Spitzensportlern gleichgestellt – denn nur diese bekommen wirklich einen gelungenen Spagat hin. Dieser Spagat kann erreicht werden durch die Kombination von RPG- und DB2-Elementen, die unabhängig von der Anweisungsform ISAM-basiert (physische oder logische Dateien) sein oder auch SQL-Komponenten in gleicher Art verwenden können (Tabellen, Indizes). Write kann sowohl für ISAM- als auch für SQL-Datenbankelemente eingesetzt werden – und zwar gleichartig!

Ähnlich verhält es sich umgekehrt – denn mit embedded SQL können sowohl ISAM-Komponenten als auch SQL-Komponenten verwendet werden. Wie das Schreiben eines Datensatzes mittels embedded SQL funktioniert, soll nachfolgend aufgezeigt werden.

3.2.7.2 Hinzufügen von neuen Datensätzen mit RPG und embedded SQL

Die Symbiose zwischen dem „klassischen“ RPG und dem „modernen“ SQL hat IBM vor einigen Jahren als sinnvolle Erweiterung für die weite Welt der System-i-Entwicklung bereitgestellt. SQL-Werkzeuge lassen sich auch mit RPG zum Schreiben von neuen Datensätzen verwenden.

Abweichend von den Voraussetzungen für das Schreiben mit Write werden keine Dateibestimmungen benötigt, denn die notwendigen Informationen zur Datei/ Tabelle werden vom Compilierprozess bereitgestellt.

Nachfolgendes Beispiel zeigt das Schreiben eines neuen Datensatzes in der Beispieldatei DemoArt.

```
d Rpg0003         pr
d p_Firm                         2
d p_Teil                        10
d p_Beze                        50

d Rpg0003         pi
d p_Firm                         2
d p_Teil                        10
d p_Beze                        50

 /FREE
  EXEC SQL
   Insert into DemoArt
          (DaFirm, DaTeil, DaBeze)
   Values (:p_Firm, :p_Teil, :p_Beze);

   *inlr = *on;

 /end-free
```

Im Vergleich zur nativen Schreibvariante mit Write fällt auf, dass der Codeausschnitt kürzer ist.

Dies liegt unter anderem an der nicht notwendigen Dateiinformation – die Dateiinformationen ermittelt der SQL Pre-Compiler – und den nicht genutzten Datenstrukturangaben. Im Beispiel wird der Datensatz in die Datei/Tabelle DemoArt auf Basis der einzelnen Felder geschrieben.

Weitere Details zur Verwendung von embedded SQL und RPG finden Sie in einem anderen Kapitel.

3.2.7.3 Löschen bestehender Datensätze mit RPG nativ

Es gibt immer wieder Situationen, in denen der Dateiinhalt bereinigt werden muss und einzelne Datensätze gelöscht werden müssen. Dies kann in RPG mit der Anweisung Delete erfolgen.

Delete erlaubt das Löschen des Datensatzes, auf dem der Verarbeitungsprozess innerhalb des RPG-Programms steht. Anders als Write beschränkt sich der Einsatz von Delete auf Datenbankdateien. Hier können alle Formen der Datenbankdateien verwendet werden, zum Beispiel physische Dateien, logische Dateien, SQL-Tabellen und SQL-Indizes.

Wie es bereits beim Schreiben neuer Datensätze ersichtlich war, muss die Datei in den Dateibestimmungen für das Verändern (in dem Fall das Löschen) zuvor definiert werden. Dazu muss in der Dateibestimmung in Spalte 17 der Wert U = Update eingegeben werden.

Zurück zur Beispieldatei DemoArt:

```
FDemoArt   uF A e           K DISK
```

Für Fans von Free-Format-RPG wird eine Datei unter anderem auch für das Löschen wie folgt definiert:

```
// Dateibestimmung
dcl-f ArtikRf Disk usage(*update) keyed;
```

Das Löschen eines Datensatzes wird mit der Anweisung Delete durchgeführt. Dabei muss beachtet werden, dass der Datensatz vor dem Löschen gesperrt wird, was mit der Leseanweisung Reade oder Chain erfolgen kann.

Nachdem der zu löschende Datensatz eingelesen und gesperrt worden ist, kann dieser mit Delete, gefolgt vom Satzformat- oder Dateinamen, gelöscht werden.

Achtung:

Löschen bedeutet wirklich löschen! Denn damit ist der Datensatz unwiederbringlich vom System entfernt.

3.2.7
Seite 14

Im nachfolgenden Beispiel wird ein typischer Chain-Zugriff im Free-Format unter Verwendung einer Schlüsseldatenstruktur verwendet. Natürlich können Sie hier jede mögliche RPG-Variante für das Lesen verwenden, also zum Beispiel einen Chain mit einer Klist-Definition, eine Reade-Anweisung mit Datenstruktur oder eine Reade-Anweisung mit Schlüsseldatenstruktur oder Klist-Definition. Details zu den unterschiedlichen Lesezugriffsformen finden Sie in einem der vorhergehenden Abschnitte.

Nun aber zum Löschen:

Nach dem Einlesen des Datensatzes und dessen Sperrung kann dieser mit Delete aus der Datei entfernt werden.

Den Programmcode finden Sie hier:

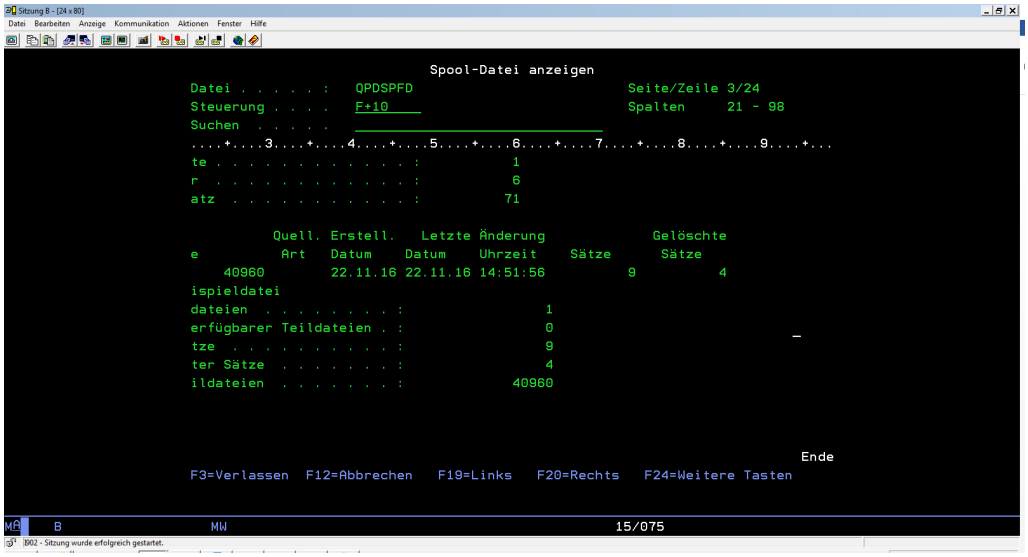
```
FDemoArt   uF A e           K DISK

d Rpg0004         pr
d p_Firm                     2
d p_Teil                    10

d Rpg0004         pi
d p_Firm                     2
d p_Teil                    10

d Arti_K        E DS                  ExtName(DemoArt:*key)
d                                     Qualified

d Arti_I        E DS                  ExtName(DemoArt:*input)
d                                     Qualified

 /Free

  // Schlüsselfelder belegen
  Arti_K.DaFirm = p_Firm;
  Arti_K.DaTeil = p_Teil;

  // Satz einlesen
  chain %kds(Arti_K:2) DemoArt Arti_I;
  if %found(DemoArt);

  // Satz löschen
  delete rDemoArt;

  endif;

  *InLR = *On;


 /end-free
```

Dieses Programm löscht den gewünschten Datensatz aus der Datei.

Dabei ist zu beachten, dass der Datensatz in der betreffenden Datei als „gelöscht" gekennzeichnet wird. Auch wenn das erst einmal nur eine Kennzeichnung ist, der Datensatz ist nun nicht mehr nutzbar und kann auch nicht wiederhergestellt werden. Als Ausnahme sei hier der Einsatz von Journalen genannt. Mit einem Journal und entsprechenden Schritten könnte der Datensatz im Notfall wiederhergestellt werden. Das aber ist ein anderes Thema.

Faktisch ist der Datensatz aber noch in der Datei enthalten. Dies ist in der Dateibeschreibung ersichtlich, denn hier ist ein Verweis auf die Anzahl der in der Datei gelöschten Datensätze zu finden.

Folgende Anzeige liefert die Information, dass sich in der Datei DemoArt derzeit vier gelöschte Datensätze befinden.



Tipp:

Wenn Sie eine größere Anzahl Datensätze in einer Datei löschen, dann sollten Sie auch dafür sorgen, dass die Datei in gewissen Zeitabständen reorganisiert wird – denn damit wird der Datensatz final aus der Datei entfernt, der belegte Plattenplatz wird freigegeben, und verschiedene Reorganisationsaufgaben werden auf Datenbankebene durchgeführt. Verwenden Sie dazu den Befehl RGZPFM. Details dazu finden Sie an anderer Stelle.

3.2.7
Seite 16

Natürlich können in einem RPG-Programm auch mehrere Datensätze gelöscht werden. Dabei werden die einzelnen Datensätze in einer Schleife eingelesen und nach dem Sperren und Lesen mittels Delete-Anweisung aus der Datei entfernt. Eine solche Leseschleife finden Sie hier:

```
d Arti_K          E DS                  ExtName(DemoArt:*key)
d                                       Qualified

d Arti_I          E DS                  ExtName(DemoArt:*input)
d                                       Qualified

 /Free

  // Schlüsselfelder belegen
  Arti_K.DaFirm = p_Firm;

  // Satz einlesen
  setll %kds(Arti_K:1) DemoArt;
  reade %kds(Arti_K:1) DemoArt Arti_I;
  dow not %eof(DemoArt);
   if Arti_I.DaAkti <> 'A';

   // Satz löschen
   delete rDemoArt;
   endif;

   reade %kds(Arti_K:1) DemoArt Arti_I;
  enddo;
```

3.2.7.4 Löschen bestehender Datensätze mit RPG und embedded SQL

Auch beim Löschen der gewünschten Datensätze müssen RPG-Entwickler wählen, ob dies mit der RPG-Anweisung Delete oder mit embedded SQL erfolgt.

Letztlich ist es reine Geschmackssache, denn der Effekt, das Löschen der Daten, ist derselbe.

Allerdings ist auch beim Einsatz von embedded SQL in RPG für das Löschen der eigentliche Code im Vergleich zur nativen RPG-Variante übersichtlicher. Das folgende Beispiel zeigt, wie ein Datensatz in der Datei DemoArt basierend auf den beiden Parametern Firma und Teilenummer gelöscht wird.

```
d Rpg0006         pr
d p_Firm                         2
d p_Teil                        10

d Rpg0006         pi
d p_Firm                         2
d p_Teil                        10

 /FREE
  Exec Sql
  set option commit = *none, datfmt = *iso, timfmt = *iso;

  EXEC SQL
   delete from DemoArt
          where DaFirm = :p_firm and DaTeil = :p_Teil;

   *inlr = *on;
```

Auch hier gilt:
Mit embedded SQL muss die Datei, deren Inhalt gelöscht werden soll, nicht in den Dateibestimmungen definiert sein. Das Löschen kann direkt für einzelne Datensätze mit der detaillierten Where-Klausel durchgeführt werden.

Insbesondere beim Löschen mehrerer Datensätze kann SQL seine Stärken im Vergleich zu RPG ausspielen, denn hier können ohne eine vorherige Leseprozedur in einer Schleife direkt alle gewünschten Datensätze mit einer Anweisung eliminiert werden.

3.2.7
Seite 18

Folgendes Codebeispiel kann dazu genutzt werden, um alle Artikelsätze aus der Datei DemoArt zu entfernen, die nicht mehr aktiv sind.

```
/FREE
 Exec Sql
 set option commit = *none, datfmt = *iso, timfmt = *iso;

 EXEC SQL
  delete from DemoArt
         where DaAkti <> 'A';

  *inlr = *on;
```

Ein vergleichbares RPG-Programm zum Löschen aller Datensätze mit Aktivkennzeichen <> ‚A' könnte folgendermaßen aussehen:

```
  FDemoArt   uF A e           K DISK

  d Rpg0008         pr
  d p_Firm                        2

  d Rpg0008         pi
  d p_Firm                        2

  d Arti_K        E DS                  ExtName(DemoArt:*key)
  d                                     Qualified

  d Arti_I        E DS                  ExtName(DemoArt:*input)
  d                                     Qualified

   /Free

    // Schlüsselfelder belegen
    Arti_K.DaFirm = p_Firm;

    // Satz einlesen
    setll %kds(Arti_K:1) DemoArt;
    reade %kds(Arti_K:1) DemoArt Arti_I;
    dow not %eof(DemoArt);
     if Arti_I.DaAkti <> 'A';

     // Satz löschen
     delete rDemoArt;
     endif;

     reade %kds(Arti_K:1) DemoArt Arti_I;
    enddo;

    *InLR = *On;
```

Vergleichen Sie selbst: Das RPG-Programm mit der nativen Delete-Anweisung ist deutlich länger als das SQL-Gegenstück. Letztlich sind jedoch beide Varianten zielführend.

3.2.7.5 Verändern bestehender Datensätze mit RPG nativ

Datensätze bestehen im Allgemeinen aus einzelnen Feldern, und es ist üblich, dass diese auch hin und wieder geändert werden. Dazu wird die RPG-Anweisung Update genutzt.

Update kann für Datenbankdateien, aber auch für Subfiles genutzt werden. In diesem Abschnitt soll sich auf den Einsatz zusammen mit Datenbankdateien konzentriert werden.

Natürlich können Datensätze nativ mit RPG-Mitteln verändert werden. Hierzu sind die Voraussetzungen auf Dateiebene zu erfüllen, wie sie auch beim Löschen definiert worden sind. So muss auf Dateiebene in der Dateiverwendung (Spalte 17) der Wert U (Update) eingegeben bzw. im Free-Format-RPG die DCL-F mit der Verwendung *Update codiert werden.

Beide Dateidefinitionsvarianten finden Sie hier:

Fix-Format-RPG:

```
h copyright('Joerg Zeig 10/2016')

FDemoArt   uf   e           K DISK
```

Free-Format-RPG:

```
//
ctl-opt copyright('Joerg Zeig 10/2016') ;

// Dateibestimmung
dcl-f DemoArt Disk usage(*update) keyed;

*inlr = *on;
```

3.2.7
Seite 20

Für das Ändern von Inhalten bestehender Datensätze sind folgende Regeln zu beachten:

1. Die Datei muss als Update-Datei in den Dateibestimmungen definiert werden.
2. Der Datensatz muss vor dem Verändern eingelesen und gesperrt werden.
3. Nach dem Lesen sollten nur noch für das Ändern relevante Aktionen mit der Datei/den Feldern durchgeführt werden.
4. Für extern beschriebene Dateien muss der Satzformatname für die Update-Anweisung angegeben werden.
5. Sollten Sie programmbeschriebene Dateiinhalte verändern wollen, dann muss der Dateiname zusammen mit der RPG-Anweisung Update angegeben werden (nicht von IBM empfohlen!).
6. Die Wertzuweisung kann im RPG direkt über die Felder oder über eine Datenstruktur erfolgen.

Allgemein wird ein Datensatz mit der RPG-Anweisung Update verändert. Dabei wird der gesamte Datensatz (alle Felder) verarbeitet. Eine Ausnahme bildet die Ergänzung der Update-Anweisung mit der Built-in Function %FIELDS. Dazu später mehr.

Ein weiteres Beispiel ist die bekannte Artikelstammdatei DemoArt.

Zuvor wurden neue Datensätze angelegt, denen noch kein Lager zugeordnet worden ist.

Hierzu einige Beispiele:

	DATEIL	DABEZE	DALAGO
01	4000	Heftstreifen	001
01	4010	Locher	001
01	4020	Kopierer	002
01	4050	Atlas 2016	001
01	5000	USB Scanner	
01	5001	USB Scanner Schwarz	
01	5002	USB Scanner gelb	

Wie zu sehen, haben die Artikel 5000, 5001 und 5002 keinen eingetragenen Lagerort (Feld DALAGO).

Ein entsprechendes RPG-Update-Programm beginnt mit der zuvor erwähnten Dateidefinition.

Nun kann entschieden werden, ob mit den Dateifeldern im RPG direkt gearbeitet oder ob eine Datenstruktur zur genauen Abgrenzung der Update-Informationen verwendet werden soll.

Das erste Beispiel zeigt die Verwendung einer Datenstruktur. Der Vorteil ist, dass durch die spezifische Definition der Datenstruktur Arti_O zur Ausgabe sichergestellt wird, dass nicht etwa ein anderer Leseprozess zugewiesene Werte verändert, was zu einem ungewollten Ergebnis führen würde.

Das Programm wird mit dem Parameter „Firma“ aufgerufen.

In den Datenstrukturdefinitionen gibt es nun drei verschiedene Datenstrukturen:

Arti_K zur Schlüsseldefinition,

Arti_I für Einleseprozesse,

Arti_O für Ausgabeprozesse.

3.2.7
Seite 22

```
FDemoArt   uf   e           K DISK

d Rpg0009         pr
d p_Firm                        2

d Rpg0009         pi
d p_Firm                        2

d Arti_K        E DS                  ExtName(DemoArt:*key)
d                                     Qualified

d Arti_I        E DS                  ExtName(DemoArt:*input)
d                                     Qualified

d Arti_O        E DS                  ExtName(DemoArt:*output)
d                                     Qualified

 /Free

  // Schlüsselfelder belegen
  Arti_K.DaFirm = p_Firm;

  // Satz einlesen
  setll %kds(Arti_K:1) DemoArt;
  reade %kds(Arti_K:1) DemoArt Arti_I;
  dow not %eof(DemoArt);
   if Arti_I.DaLago = *blank;
   Arti_O = Arti_I;
   Arti_O.DaLago = '001';

   // Satz fortschreiben
   update rDemoArt Arti_O;
   endif;

   reade %kds(Arti_K:1) DemoArt Arti_I;
  enddo;

  *InLR = *On;
```

Mit dem Parameter „Firma“ werden nun alle Datensätze der Datei DemoArt eingelesen. Dazu wird zunächst der Positionierer SETLL verwendet, und dann werden mit dem Schlüssel „Firma“ (das ist das erste Schlüsselfeld in der Datei) alle Datensätze aus der Datei DemoArt ausgelesen. Das Ergebnis wird automatisch in die Einlesedatenstruktur Arti_I gestellt, die denselben Aufbau hat wie die zugrunde liegende Datei.

Da die Datenstruktur als qualifizierte Datenstruktur definiert wurde, werden demzufolge die Felder im RPG-Programm auch qualifiziert angesprochen. Um zu prüfen, ob der Inhalt des Lagerorts (Feld DALAGO) leer ist, muss der Inhalt von Arti_I.DALAGO abgefragt werden.

```
if Arti_I.DaLago = *blank;
```

Jetzt wird sichergestellt, dass die eingelesenen Werte nicht verändert werden. Dazu muss der gesamte Inhalt der Datenstruktur Arti_I der Ausgabedatenstruktur Arti_O zugewiesen werden.

```
Arti_O = Arti_I;
```

Die Datenstruktur Arti_O soll auch die Ausgabedatenstruktur für die Update-Anweisung sein und muss folglich in den Definitionsbestimmungen mit *Output definiert werden.

Mit der Bedingungsprüfung IF werden nun alle Datensätze, deren Feldinhalt „Lagerort“ leer ist, verarbeitet.

Als Nächstes wird der Ausgabedatenstruktur oder besser dem Feld der Ausgabedatenstruktur Arti_O.DALAGO der gewünschte Zielwert zugewiesen.

```
Arti_O.DaLago = '001
```

Damit hat das Feld die geforderte Wertzuweisung erhalten und muss jetzt in der Datei fortgeschrieben werden.

Dazu wird die RPG-Anweisung Update zusammen mit dem Satzformatnamen und der Ausgabedatenstruktur Arti_O genutzt.

```
// Satz fortschreiben
update rDemoArt Arti_O;
```

Auch hier gilt wie bereits bei der Neuanlage von Datensätzen, dass Sie als Anwendungsentwickler sicherstellen müssen, dass die Anforderungen an den Datensatz auch erfüllt werden. Das bedeutet unter anderem die korrekte Wertzuweisung, die korrekte Verwendung von Datentypen sowie die Beachtung von Schlüsselfeldern mit den damit verbundenen Einschränkungen und Besonderheiten. Auch dies sind keine RPG-Spezifika, sondern es ist eine wichtige Programmierrichtlinie!

Auf diese Weise werden im Programm alle Datensätze der Firma 01 verarbeitet, die keinen Lagerort eingetragen haben.

Achtung:

Update schreibt den gesamten Datensatz fort, also alle Felder bzw. Spalten.

3.2.7
Seite 24

Es kann durchaus sinnvoll sein, gezielt die Werte einzelner Felder zu verändern. In einem solchen Fall kann eine Sonderform von Update in Kombination mit der BIF %FIELDS genutzt werden. „Sinnvoll“ bedeutet in diesem Zusammenhang Zeiteinsparung, denn nur die definierten Felder werden über den Eingabe-/Ausgabe-Buffer verarbeitet. Bei großen Datenmengen kann sich das durchaus bemerkbar machen!

Hier ein Beispiel für den Einsatz von %Fields:

```
// Satz fortschreiben
update rDemoArt %Fields(DaLago);
```

Anmerkung:

%Fields steht nur im Free-Format zur Verfügung. Hier gibt es keine Fix-Format-Variante.

3.2.7.6 Verändern bestehender Datensätze mit embedded SQL

Neben den nativen RPG-Update-Möglichkeiten können RPG-Entwickler auch embedded SQL für das Fortschreiben von bestehenden Datensätzen nutzen. Auch hier sind das blockweise Verarbeiten mehrerer Datensätze und die Nutzung der SQL-Syntax von Vorteil.

Im Folgenden wird exemplarisch das Ändern bestehender Datensätze in der Beispieldatei DemoArt aufgezeigt.

Auch hier sollen alle Datensätze, die keinen Lagerort beinhalten, mit dem Standardlagerort ‚01' versehen werden. Ein RPG-Beispiel dazu finden Sie hier:

```
/FREE
 Exec Sql
 set option commit = *none, datfmt = *iso, timfmt = *iso;

 EXEC SQL
  update DemoArt
        set DaLago = '01'
  where DaLago = ' ';

  *inlr = *on;
```

3.2.7
Seite 26

3.2.8 Built-in Functions

RPG wird oft als „alte“ Sprache angesehen, doch ist man geteilter Meinung hinsichtlich dessen, was in diesem Zusammenhang unter „alt“ verstanden wird. Sicher ist jedenfalls eines: Seit seiner Einführung hat sich RPG extrem gewandelt und weiterentwickelt.

Vergleicht man heute die Möglichkeiten, die Free-Format-RPG bieten, mit den frühen RPG-Varianten im Fix-Format, dann haben die beiden Varianten nur noch sehr wenig gemeinsam. Doch sind es nicht nur die Neuerungen rund um die Free-Format-RPG-Version, die uns für Anwendungsentwickler Erleichterungen darstellen, sondern es sind viele Funktionen, die auch im Fix-Format-Bereich vorzufinden sind. Einige Neuerungen im RPG in den letzten Jahren betreffen die Verwendung und Bereitstellung der sogenannten Built-in Functions, worunter sich Funktionen verbergen, mit denen „Standard“-Anforderungen relativ einfach umgesetzt werden können.

RPG IV und die Free-Format-Formen von RPG bieten eine Vielzahl an Built-in Functions, von denen einige in direktem Bezug mit der Datenmanipulation stehen. Mit Hilfe von verschiedenen Built-in Functions können unter anderem Konvertierungen von Datentypen recht einfach durchgeführt, Substrings erstellt und verarbeitet sowie Felder bzw. Feldinhalte verbunden werden. Wenn Sie zum Beispiel den Inhalt eines Feldes mit einem bestimmten Datentyp in einen anderen Datentyp konvertieren wollen, dann stehen Ihnen leistungsfähige Built-in Functions zur Verfügung, die Ihnen die Realisierung der Anforderung vereinfachen.

Im nachfolgenden Kapitel sollen einige der verfügbaren Built-in Functions im Detail vorgestellt werden.

3.2.8
Seite 2

Untenstehende Tabelle gibt einen Überblick über die wichtigsten Built-in Functions.

Built-in Functions

Built-in Function	Beschreibung
%Char	Konvertieren in ein Zeichenfeld
%Check	Prüfen von Zeichenfeldern
%Checkr	Rückwärtiges Prüfen
%Date	Umwandeln ins Datumsformat
%Days	Anzahl Tage
%Dec	Umwandeln in ein gepacktes dezimales Format
%Dech	Umwandeln in ein gepacktes dezimales Format mit Rundung
%Decpos	Abfragen der Dezimalpositionen
%Diff	Differenz zwischen zwei Datums-, Zeit- oder Timestamp-Feldern
%Editc	Editieren eines Wertes mit einem Editier-Code
%Elem	Abrufen der Anzahl der Elemente
%Hours	Anzahl der Stunden
%Int	Umwandeln in ein Integer-Format
%Inth	Umwandeln in ein Integer-Format mit Rundung
%Len	Abrufen oder Setzen der Länge
%Minutes	Anzahl in Minuten
%Month	Anzahl in Monaten
%Mseconds	Anzahl in Millisekunden
%Replace	Ersetzen der Zeichenfolge
%Scan	Suchen der Zeichen
%ScanR	Rückwärtiges Suchen der Zeichen
%ScanRpl	Suchen und Ersetzen der Zeichen
%Seconds	Anzahl in Sekunden
%Size	Größenabfrage in Bytes
%Subdt	Extrahieren eines Datums-, Zeit- oder Timestamp-Bereichs
%Subst	Abrufen eines Substring (Unterzeichenfolge)
%Time	Umwandeln in ein Zeitformat
%Timestamp	Umwandeln in ein Timestamp-Format
%Trim	Abschneiden einer Zeichenfolge
%Triml	Abschneiden der führenden Zeichen
%Trimr	Abschneiden der anhängenden Zeichen
%Uns	Umwandeln in ein unsigniertes Format
%Unsh	Umwandeln in ein unsigniertes Format mit Rundung
%Xlate	Übersetzen
%Years	Anzahl in Jahren

3.2.8.1 Umwandlung vom dezimalen Zeichenformat in Character

Es kommt immer wieder vor, dass zu verarbeitende Daten nicht im gewünschten Format vorliegen. Dies betrifft vor allem die unterschiedlichen Möglichkeiten, wie ein Datumswert definiert wurde, oder die „interessante“ Definition einer „Nummer“ als alphanumerischen Wert.

Bevor IBM die Möglichkeit der Datentypumwandlung mit Hilfe der verschiedenen Built-in Functions geboten hat, konnten Umsetzungen von Dezimalfeldern in Zeichenfelder mit Hilfe von Move-Anweisungen (Move und Movel) durchgeführt werden. Diese Anweisung ist im Free-Format-RPG IV verloren gegangen. Der Versuch einer Umsetzung mit der Anweisung EVAL führt häufig zu Kompilierfehlern – und das mit Recht. Denn es kann nicht wirklich sichergestellt werden, dass die Konvertierung von einem Datenformat ins andere korrekt durchgeführt wird. Hier hilft die Built-in Function %Char, die genau diese Aufgabe übernimmt.

Mit Hilfe der %Char-Funktion kann ein numerischer Wert in eine Zeichenfolge umgesetzt werden.

Der allgemeine Aufbau einer solchen %Char-Funktion sieht wie folgt aus:

```
dcl-s Char_Var1 char (100);
dcl-s Char_Num1 zoned (15:2);

Char_Num1 = 00123,45;
char_var1 = %char(Char_Num1);

*inlr = *on;
```

Built-in Function %Char

Beim Betrachten der Ausführung des Ergebnisses der Variablen Char_Var1 ist der Wert DSPLY 123.45 zu sehen. Es fällt dabei auf, dass die führenden Nullen im Ergebnisfeld nicht enthalten sind. Diese Besonderheit zeichnet die Built-in Function %Char aus.

Allerdings bietet IBM andere Built-in Functions an, mit denen die aus dem nativen RPG bekannten Editierfunktionen auch in einer Built-in Function genutzt werden können.

Eine Eigenart der %Char-Funktion, der sich in diesem Zusammenhang zunutze gemacht werden kann, ist die Umwandlung von führenden Null-Zeichen in Leerstellen eines Zeichenfeldes. Sollte der Effekt, dass das Zeichenfeld anstelle führender Leerstellen führende Nullen beinhaltet, nicht gewünscht sein, dann muss die Built-in Function %Editc in Verbindung mit dem x-Edit-Code verwendet werden. Nachfolgend finden Sie die allgemeine Aufbauform der Built-in Function %Editc.

3.2.8
Seite 4

```
dcl-s Char_Var1 char (50);
dcl-s Char_Num1 zoned (15:2);

Char_Num1 = 00123,45;
char_var1 = %editc(Char_Num1 : 'X');

dsply Char_Var1;

*inlr = *on;
```

Built-in Function %Editc

Das Ergebnis dieses Code-Beispiels würde so aussehen: 000000000012345.

Natürlich lassen sich auch numerische Werte mit Hilfe von Built-in Functions in Zeichenfelder umwandeln. Ein Beispiel dazu finden Sie hier:

```
dcl-s Ergebnis char (50);
dcl-s Num1 zoned (7:2) inz(4711,55);
dcl-s Num2 zoned (5:0) inz(00312);

// Umsetzung ohne führende Null Ergebnis: 4711.55
Ergebnis = %char(Num1);
dsply Ergebnis;

// Umsetzung mit führender Null Ergebnis: 312
Ergebnis = %char(Num2);
dsply Ergebnis;

// Umsetzung mit führender Null mit %editc Ergebnis: 00312
Ergebnis = %editc(Num2:‘X‘);
dsply Ergebnis;

*inlr = *on;
```

Umwandlung numerischer Felder in Zeichenfelder mit den Funktionen %Char und %Editc

3.2.8.2 Umwandlung von Zeichenfeldern in gepackte dezimale Daten

Bei der Umsetzung einer Zeichenfolge in ein numerisches Feld kann unter Umständen davon ausgegangen werden, dass sowohl Dezimaltrennzeichen als auch Vorzeichen in der Zeichenkette enthalten sind. Die Built-in Function %Dec verfügt über drei Parameter, die für die Umsetzung eines Zeichenfeldes in ein gepacktes dezimales Feld verwendet werden können.

Der allgemeine Aufbau der Built-in Function %Dec sieht wie folgt aus:

```
dcl-s Eingabe char (20) inz('1234') ;
dcl-s Num1 packed (7:0) ;

// Umsetzung Ergebnis: 1234
Num1 = %dec(Eingabe : 4:0);
dsply Num1;

*inlr = *on;
```

Built-in Function %Dec

Beim ersten Parameter dieser Funktion handelt es sich um die umzusetzende Zeichenfolge. Es kann sich dabei um ein Datumsfeld, ein Zeitfeld, ein Feld vom Typ Timestamp oder ein Zeichenfeld handeln. In der Zeichenfolge können sich am Anfang oder am Ende auch Pluszeichen (+) oder Minuszeichen (–) befinden. Außerdem dürfen dort auch Dezimaltrennzeichen vorkommen. Vor einer Umsetzung der Zeichenfolge in ein numerisches Feld muss allerdings sichergestellt werden, dass eventuell in der Zeichenfolge enthaltene Währungszeichen und Trennzeichen für 1000er-Schritte aus der Zeichenfolge entfernt wurden. Das Entfernen dieser Zeichen kann mit Hilfe der Built-in Function %Xlate als Erweiterung von %Dec leicht durchgeführt werden.

Der zweite Parameter der Funktion %Dec definiert die Länge des Rückgabewertes, und mit dem dritten wird die Anzahl der Dezimalstellen im Rückgabewert festgelegt. Bei diesen beiden Parametern kann es sich wahlweise um Literale oder benannte Konstanten handeln. Hier können allerdings keine weitere Built-in Function verwendet werden.

Die verschiedenen Möglichkeiten sind in der nachfolgenden Abbildung ersichtlich. Beachten Sie dort auch die Verwendung von zwei zusätzlichen Built-in Functions (%Len und %Decpos). Mit diesen beiden Funktionen werden die Länge des numerischen Ergebnisfeldes und die Anzahl der Dezimalstellen des Feldes abgerufen.

3.2.8
Seite 6

```
dcl-s Char1  char (20) inz('1234,55');
dcl-s Char2  char (10) inz('-12 34 5 6');
dcl-s Ergebnis zoned (9:2) ;
dcl-c laenge %len(Ergebnis) ;
dcl-c laenge2 %decpos(Ergebnis) ;

// Umsetzung Char1 in Dec mit %dec Ergebnis: 123455
Ergebnis = %dec(Char1 : 9:2);
dsply Ergebnis;

// Umsetzung Char2 in Dec mit %dec Ergebnis: 12345600-
Ergebnis = %dec(Char2 : 9:2);
dsply Ergebnis;

Ergebnis = %dec(Char2 : laenge + laenge2:0)/10**laenge2;
dsply Ergebnis;

*inlr = *on;
```

Verwendung der Built-in Function %Dec zur Umsetzung eines Zeichenfeldes in ein dezimales Feld

3.2.8.3 Teilzeichenfolgen (Substrings)

Mit der Built-in Function %Subst können unterschiedliche Teilzeichenfolgen aus einer Zeichenkette extrahiert werden. Diese Funktion ist vergleichbar mit der CL-Funktion %Sst und ebenfalls im Funktionsumfang gleich mit der RPG-Anweisung „Subst“ (Substring). Der Rückgabewert einer solchen %Subst-Funktion ist immer ein Zeichenwert.

Die Built-in Function %Subst verfügt über drei Parameter: eine Suchzeichenfolge, einen Startpunkt und eine Länge. Sollten zu wenige Zeichen in einer Zeichenkette vorhanden sein, die aufgrund des definierten Startpunkts bzw. der festgelegten Länge zu einem Fehler führen, dann zeigt der Kompiler (oder im Fall des Einsatzes von RDi der Editor) diesen Fehler entsprechend an.

Um aus einer vorgegebenen Zeichenfolge eine Teilzeichenfolge zu erstellen, muss %Subst eingegeben werden (siehe nachfolgende Abbildung).

```
dcl-s String1 char (11) inz('Hans Willi');
dcl-s String2 char (6) inz('Winzig');
dcl-s Ergebnis char (15) inz;

// Umsetzung ab 2. Stelle Ergebnis: 'Willi'
Ergebnis = %subst(String1 : 6:5);
dsply Ergebnis;

// Verketten mit SUBST Ergebnis: "WilliWinzig'
%subst(Ergebnis:6:%len(String2)) = String2;
dsply Ergebnis;

*inlr = *on;
```

Extrahieren von Teilzeichenfolgen mit Hilfe der Built-in Function %Subst

3.2.8.4 Ersetzen von Zeichenfolgen

Die sehr leistungsfähige Built-in Function %Replace hat keine in RPG oder CL vorhandene vergleichbare Funktion. Sie verfügt über vier Parameter und kann sehr flexibel eingesetzt werden, und mit deren Hilfe können beispielsweise Zeichen innerhalb einer Zeichenfolge ersetzt, Zeichen in eine Zeichenkette eingefügt oder auch Zeichen aus einer Zeichenfolge gelöscht werden. Der wesentliche Unterschied zwischen %Subst und %Replace liegt darin, dass %Replace zwei Zeichenfolgenoperanten verwendet: „Von“ und „Bis“.

Der dritte Parameter der %Replace-Funktion ist der Startpunkt für den „To“-Operanten, und beim vierten handelt es sich um die Länge.

Die Angabe einer Länge „0“ für den vierten Parameter hat das Aktivieren der Einfügen-Funktion zur Folge.

Das Löschen von Zeichen wird mit Hilfe zweier aufeinanderfolgender einfacher Anführungszeichen (’’) als ersten Operanten initiiert.

In der folgenden Abbildung finden sich einige Beispiele für die Built-in Function %Replace.

```
dcl-s String1 char (20) inz('Wohnort Köln');
dcl-s String2 char (5) inz('Koeln');
dcl-s Ergebnis char (20) inz;

// Ersetzen Ergebnis: 'Wohnort Koeln'
Ergebnis = %replace(String2: String1:9:%len(String2));
dsply Ergebnis;

// Ersetzen Ergebnis: 'Wohnort: Koeln'
Ergebnis = %replace(':' : Ergebnis:8:1);
dsply Ergebnis;

*inlr = *on;
```

Verwendung der Built-in Function %Replace

3.2.8.5 Suchen und Ersetzen von Zeichenfolgen

Dass die IBM auch das gute alte RPG des Öfteren funktional erweitert, zeigt die relativ neue Built-in Function %ScanRPL, mit der sich Zeichenfolgen ermitteln und durch einen vorgegebenen Ersatzwert austauschen lassen. Damit hat IBM quasi eine Kombination aus den beiden zuvor genannten Built-in Functions Suchen (%Scan) und Ersetzen (%Replace) geschaffen.

3.2.8.6 Verknüpfen von Zeichenketten und Entfernen von Leerstellen

In Free-Format-RPG IV wird der Operant „+“ als Ersatz für den aus Fix-Format-RPG bekannten Operanten „CAT“ verwendet. Mit Hilfe des Operanten „+“ können lediglich Zeichenketten miteinander verknüpft werden. Einer der wesentlichen Unterschiede im Vergleich zur Anweisung „CAT“ liegt darin, dass keine Leerstellen abgeschnitten werden. (Sollte das gewünscht sein, dann bietet sich die Kombination mehrerer Built-in Functions an – im Beispiel die Built-in Function %Trim, mit Hilfe derer Leerzeichen auf drei verschiedene Weisen entfernt werden können:

- %Trim entfernt sowohl führende als auch nachfolgende Leerstellen.
- %Triml entfernt führende Leerstellen.
- %Trimr entfernt nachfolgende Leerstellen.

Jede dieser drei Built-in Functions entfernt bei der Ausführung die Leerstellen Schritt für Schritt. Auf diese Weise wird eine Eingabe-Zeichenfolge so umgesetzt, dass darin enthaltene Leerstellen nach den jeweiligen Vorgaben vollständig entfernt werden. Sollten sich innerhalb einer zu verarbeitenden Zeichenkette eingeschlossene Leerzeichen befinden, zum Beispiel „Bad Mergentheim“, dann werden diese nicht entfernt.

Nachfolgende Abbildung zeigt einige Beispiele für das Verketten von Zeichenfolgen und das Entfernen von Leerstellen.

```
dcl-s Stadt  char (20) inz('Frankfurt');
dcl-s PLZ     char (5) inz('60000');
dcl-s Ergebnis char (30) inz;

// Ersetzen Ergebnis: '60000 Frankfurt'
Ergebnis = %trim(PLZ) + ' ' + %trimr(Stadt);
dsply Ergebnis;

*inlr = *on;
```

Zeichenketten zusammenfügen und Leerstellen entfernen

3.2.8.7 Umwandlung von Zeichen in Integer-Format

Eine weitere, sehr hilfreiche Built-in Function ist %Int, mit der ebenfalls eine Datentypumsetzung durchgeführt werden kann. Der eigentliche Einsatzbereich für die %Int-Funktion (und die Funktion %Inth) war das Extrahieren eines gesamten ganzzahligen Bereichs aus einer gemischten Zahl. Die Built-in Function %Inth stellt eine Alternative zur Built-in Function %Int dar, allerdings mit einer Rundungsfunktion.

Zudem verfügt die Built-in Function %Int über die Möglichkeit, dass als Parameter wahlweise eine Zeichenfolge oder ein Ausdruck angegeben werden kann. Die Zeichenfolge kann dabei ein Plus- oder Minuszeichen beinhalten, das entweder am Anfang oder am Ende der Zeichenfolge stehen muss. Des Weiteren darf in der Zeichenfolge auch ein Dezimalkomma oder ein Dezimalpunkt enthalten sein. Leerstellen, die sich an jeder beliebigen Stelle innerhalb der Zeichenfolge befinden, werden ignoriert.

Mit Hilfe der Built-in Functions %Int und %Inth können auch ganzzahlige Bereiche aus einem Gleitkommabereich extrahiert werden. Allerdings kann keine Gleitkommakonstante verwendet werden (zum Beispiel 7,4E3).

Das Ergebnis einer %Int- bzw. %Inth-Funktion kann überall in einem Programm verwendet werden, wo ein Feldtyp „integer“ erwartet wird – in Feldgruppenindizes, Längenwerten oder Schleifenparametern, um nur einige zu nennen. Aber seien Sie vorsichtig: Das Ergebnis kann unter Umständen auch ein Negativ-Integer sein!

Eine weitere Built-in Function, die durchaus vergleichbar mit %Int ist, doch nur in Verbindung mit positiven Integer-Werten eingesetzt werden kann, ist %Uns. Auch hierzu gibt es eine weitere Variante, mit der Rundungen durchgeführt werden können – die Built-in Function %Unsh. Wie auch bei der Built-in Function %Int, werden bei %Uns alle Dezimalstellen abgeschnitten.

Sowohl %Uns als auch %Unsh bieten sich gerade im Zusammenhang mit Feldgruppenindizes oder Schleifenparametern an, denn hier ist das Ergebnis in jedem Fall ein positiver Wert. In nachfolgender Abbildung werden einige Beispiele für die Built-in Functions %Int, %Inth, %Uns und %Unsh gezeigt.

```
dcl-s MixNum   zoned(7:2) inz(4711,66);
dcl-s CharNum  char(10)   inz('-1317,78');
dcl-s Ergebnis zoned(10:0) ;

// %int ruft ganzzahligen Wert ab: Ergebnis: 4711
Ergebnis = %int(MixNum);
dsply Ergebnis;

// %inth mit Umsetzung und Rundung Ergebnis: 4712
Ergebnis = %inth(MixNum);
dsply Ergebnis;

// %int mit Umsetzung und Rundung Ergebnis: 1317-
Ergebnis = %int(CharNum);
dsply Ergebnis;

// %inth mit Umsetzung und Rundung Ergebnis: 1318-
Ergebnis = %inth(CharNum);
dsply Ergebnis;

// %uns ruft ganzzahligen Wert ab: Ergebnis: 4711
Ergebnis = %uns(MixNum);
dsply Ergebnis;

// %unsh mit Umsetzung und Rundung Ergebnis: 4712
Ergebnis = %unsh(MixNum);
dsply Ergebnis;

*inlr = *on;
```

Verwendung der Built-in Functions %Int, %Inth, %Uns und %Unsh

Ergänzung 45/2018

3.2.8.8 Ermitteln von Größe, Länge und Dezimalstellen

Der Einsatz der Built-in Functions %Size, %Len und %Decpos ist recht weit verbreitet und wird gerade in Kombination mit anderen Built-in Functions häufig genutzt. Free-Format-RPG-Entwickler machen in der Regel häufig Gebrauch von diesen Funktionen. Das liegt nicht nur an ihrer Handlichkeit, sondern auch an der Möglichkeit, Fehler von vornherein zu vermeiden.

Die Built-in Function %Size liefert als Rückgabewert die Anzahl der Bytes des ersten Arguments zurück. Ein zweiter Operant, „*ALL“, kann in Verbindung mit Datenbereichen oder Datenstrukturen eingesetzt werden, um die Gesamtgröße der in diesen Elementen enthaltenen Teilbereiche zu ermitteln. Bitte beachten Sie, dass die Built-in Function %Size die Anzahl der Bytes, nicht aber die Länge ermittelt. Dieser Unterschied wirkt sich nicht bei Character-Feldern aus. Für diese ist der Rückgabewert von „Size“ und „Length“ gleich. Anders verhält sich das allerdings für die meisten numerischen Variablen, bei denen es zum Teil erhebliche Unterschiede in der Anzahl der Bytes und der eigentlichen Länge gibt.

Die Built-in Function %Len verwendet lediglich einen Parameter, der wahlweise eine Variable oder ein Ausdruck sein kann. In beiden Fällen wird die Länge zurückgegeben. Bei Character-Feldern ist die Länge gleich der Größe in Bytes. Für numerische Felder ist dies die Anzahl der Ziffern. Es gibt weitere Built-in Functions, wie zum Beispiel %Subst und %Replace, die häufig auch die Länge der Variablen benötigen.

Die Built-in Function %Len kann auch in den Fällen verwendet werden, bei denen eine For-Gruppe eingesetzt wird. Hier lässt sich mit Hilfe von %Len der Stopppunkt ermitteln, wenn die Variable einen bestimmten Wert annimmt. Der Einsatz von %Len bietet die Flexibilität, anders als bei der Verwendung von Konstanten, den Inhalt von Variablen oder Parametern automatisch zu ändern. Das hat den Vorteil, dass sich einiger Codieraufwand gespart werden kann, wenn sich zum Beispiel die Länge einer Variablen verändert.

Im Zusammenhang mit der Built-in Function %Decpos wird ein numerischer Parameter (kein Fließkomma) benötigt. Als Rückgabewert liefert diese Funktion die Anzahl der Dezimalstellen des Parameters als Integer-Wert. Dabei kann der Parameter auch ein Ausdruck sein. Fließkommavariablen werden allerdings nicht unterstützt.

In folgender Abbildung werden Beispiele für die Built-in Functions %Size, %Len und %Decpos gezeigt.

```
dcl-s Array    char(8)   Dim(1000) based(ptr);
dcl-s Char     char(10)   inz('-1317 7 8');
dcl-s Ergebnis zoned(9:2) ;
dcl-c LngA     %len(Ergebnis);
dcl-c DecA     %decpos(Ergebnis);

// Speicher für 100 Elemente zuweisen
ptr = %alloc(%size(Array) * 100);

// %dec Umsetzung Ergebnis: 1317.77-
Ergebnis = %dec(Char: LngA + DecA:0)/10**DecA;
dsply Ergebnis;

*inlr = *on;
```

Verwendung der Built-in Functions %Size, %Len und %Decpos

3.2.8.9 Anzahl der Elemente

Die Built-in Function %Elem wird immer dann benötigt, wenn die Anzahl der einzelnen Elemente von Tabellen, Datenbereichen oder Datenstrukturen ermittelt werden muss. Diese Funktion kann entweder als Wert in einem Definitionsspezifikationsschlüsselwort angegeben oder als Faktor in einer Rechenoperation codiert werden.

Wie die Built-in Functions %Len und *Like kann auch die Built-in Function %Elem bei der Vermeidung bzw. Reduzierung von potenziellen Fehlern helfen, die auch bei zukünftigen Programmänderungen auftreten können (zum Beispiel als Kontrolle von Schleifen). Sobald sich die Anzahl der Elemente in der Definition eines Elements ändert, ändert sich auch der Wert von %Elem in allen Bereichen des Programms.

Der Einsatz von %Elem zur Kontrolle von For-Schleifen und anderen Schleifenkonstrukten ist ein Beitrag zu einem guten Programmierstil.

Nachfolgende Abbildung zeigt die Verwendung der Built-in Function %Elem.

```
dcl-s Array1   char(8)   Dim(50);
dcl-s Array2   char(20)  Dim(%elem(Array1));
dcl-s z        zoned(2:0) ;

for z = 1 to %elem(Array1);
   Array2(z) = %subst(Array1(z):4:3);
endfor;

*inlr = *on;
```

Verwendung der Built-in Function %Elem

3.2.8.10 Suchfunktionen

Es gibt drei verschiedene Built-in Functions, die Sie bei der Suche nach bestimmten Zeichen oder auch Zeichenketten innerhalb eines Character-Feldes unterstützen. Die Built-in Function %Check arbeitet auf dieselbe Weise, wie es bereits von der RPG-Anweisung „Check“ bekannt ist.

Die Built-in Function %Check benötigt als ersten Parameter ein oder mehrere Zeichen, die als Literal oder benannte Konstanten angegeben werden können. Diese Zeichen-„Liste“ wird auch „Vergleicher“ genannt. Beim zweiten Parameter handelt es sich um die zu durchsuchende Zeichenkette. Der dritte, optionale Parameter legt fest, an welcher Stelle innerhalb des zweiten Parameters nach der Übereinstimmung der Zeichen gesucht werden soll. Wenn der dritte Parameter nicht angegeben wird, dann wird als Standardstartposition „1“ angenommen.

Die Built-in Function %Check arbeitet auf die Weise, dass der zu durchsuchende Bereich (Parameter 2) so lange weiterverarbeitet wird, wie keine Übereinstimmung mit dem vergleichenden Wert (Parameter 1) gefunden wird. Dieser Vergleichsvorgang wird bis zum Ende des zweiten Parameters durchgeführt. Wird eine Übereinstimmung gefunden, dann enthält der Rückgabewert die Position innerhalb des zweiten Parameters, in dem die Übereinstimmung lokalisiert wurde. Wurde keine Übereinstimmung gefunden, dann lautet der Rückgabewert „0“.

Die Built-in Function %Checkr ist das Äquivalent zur bekannten RPG-Anweisung CheckR (Rückwärts prüfen). Bei dieser Funktion kommen dieselben Parameter zum Einsatz, die Sie bereits bei der Built-in Function %Check kennengelernt haben. Der einzige Unterschied liegt darin, dass der Startpunkt als Default-Wert nicht die erste Stelle innerhalb des zu durchsuchenden Bereichs annimmt, sondern die letzte Stelle. Die Suche wird dann von rechts nach links durchgeführt. Dieser Vorgang wird so lange wiederholt, bis ein Zeichen gefunden wurde, das sich nicht in der Suchliste befindet, oder aber das Ende des zu durchsuchenden Bereichs erreicht wurde.

Bei %Scan handelt es sich um eine weitere Built-in Function, die als Ersatz für einen bisher vorhandenen Operanten „Scan“ zum Einsatz kommt. Der erste Parameter dieser Funktion ist das Suchargument. Dabei handelt es sich um eine Zeichenfolge, die aus einem oder mehreren Zeichen bestehen kann. Hier ist die Verwendung eines Literals, einer benannten Konstanten oder einer Variablen möglich. Der zu durchsuchende String befindet sich im zweiten Parameter. Beim dritten Parameter handelt es sich wieder um einen optionalen Parameter, mit dem die Startposition innerhalb des zweiten Parameters festgelegt werden kann. Ist der dritte Parameter nicht angegeben, dann wird als Standardstartposition „1“ angenommen.

3.2.8
Seite 16

Die %Scan-Funktion durchsucht mit Hilfe des ersten Parameters den Inhalt des zweiten Parameters. Wird eine erste genaue Übereinstimmung gefunden, dann wird als Rückgabewert die Anfangsposition der Übereinstimmung innerhalb des zweiten Parameters zurückgegeben. Kann keine Übereinstimmung festgestellt werden, dann lautet der Rückgabewert der Built-in Function %Scan „0“.

Ein Merkmal des Operation Code „Scan“ ist die Möglichkeit, mehrere Übereinstimmungen in einen Datenbereich zurückzugeben. Diese Möglichkeit fehlt jedoch bei der Built-in Function %Scan.

In nachfolgender Abbildung finden Sie Beispiele für die Built-in Functions %Check, %Checkr und %Scan.

```
dcl-s String1  char(20)  inz(' Bad Füssing ');
dcl-s String2  char(20)  inz('Dortmund     ');
dcl-s Ergebnis zoned(2:0)  inz;

// Stellen ohne Blank finden - Ergebnis: 2
Ergebnis = %check(' ' : String1);
dsply Ergebnis;

// letzte Stelle ohne Blank suchen  Ergebnis: 8
Ergebnis = %checkr(' ' : String2);
dsply Ergebnis;

// nach rtm suchen - Ergebnis: 3
Ergebnis = %scan('rtm' : String2);
dsply Ergebnis;

*inlr = *on;
```

Beispiele für die Built-in Functions %Check, %Checkr und %Scan

3.2.8.11 Zeichenfolgen umsetzen

Eine weitere Art der Datenmanipulation in RPG ist die Umsetzung von Zeichen innerhalb einer Zeichenkette. Dabei werden die Zeichen einzeln nacheinander umgesetzt. Die Built-in Function %Xlate ersetzt die RPG-Anweisung „Xlate".

Der erste Parameter der Built-in Function %Xlate ist die „Von"-Zeichenkette. Als „Nach"-Zeichenkette ist der zweite Parameter zu verwenden. Diese Funktion führt eine Schritt-für-Schritt-Umsetzung durch. Beim dritten Parameter handelt es sich um die umzusetzende Zeichenkette. Es kann optional auch noch ein vierter Parameter angegeben werden, dessen Default-Wert „1" verwendet wird, wenn dieser Parameter nicht definiert ist. Beim optionalen Parameter 4 handelt es sich um die Startposition für die Umsetzung.

Die Built-in Function %Xlate arbeitet auf folgende Weise:

Das erste Zeichen aus der „Von"-Zeichenkette wird mit jedem Zeichen in der „Nach"-Zeichenkette verglichen. Wird eine Übereinstimmung gefunden, dann wird der Parameter in der „Nach"-Zeichenkette, der mit derselben relativen Position in der „Von"-Zeichenkette gefunden wurde, ersetzt. Das Resultat ersetzt dann das Zeichen der umzusetzenden Zeichenkette im dritten Parameter. Wird keine Übereinstimmung zwischen dem ersten und dem zweiten Parameter gefunden, dann findet kein Austausch des Zeichens im dritten Parameter statt.

Die Verarbeitung erfolgt Zeichen für Zeichen, und zwar von links nach rechts, und wird so lange fortgesetzt, bis das Ende der zu verarbeitenden Zeichenkette erreicht wird.

In nachfolgender Abbildung ist ein Beispiel für die Built-in Function %Xlate zu sehen.

```
dcl-s Preis    char(20)  inz('Eur 4.711,88');
dcl-s Char     char(4)   inz('Eur');
dcl-s Ergebnis zoned(9:2)  inz;

// Stellen ohne Blank finden - Ergebnis: 2
Ergebnis = %dec(%xlate(Char: '    ' : Preis):9:2);
dsply Ergebnis;

*inlr = *on;
```

Verwendung der Built-in Function %Xlate

Ergänzung 45/2018

3.2.8.12 Kombinieren von Built-in Functions

Die Verwendung einer Built-in Function in Kombination mit einer weiteren Built-in Function erscheint auf den ersten Blick ein wenig heikel. RPG-Programmierer wissen, dass jeder Verarbeitungsschritt so gestaltet sein muss, dass dieser einzeln, Schritt für Schritt, Zeile für Zeile codiert werden muss. Aber das ist lediglich ein Grundkonzept, das bei der Einführung der ersten RPG-Versionen in den frühen 1960er-Jahren Bestand hatte. Diese Art der Programmierung hat sich spätestens mit der Verfügbarkeit des erweiterten Faktors 2 grundlegend geändert. Heute können innerhalb einer Programmzeile gleichzeitig mehrere Anweisungen, Vergleiche etc. angegeben werden.

Wie es anhand der Beispiele gezeigt wurde, verwenden Built-in Functions häufig Parameter, die als Rückgabewert verwendet werden und als Eingangsparameter einer weiteren Built-in Function dienen können. Es ist durchaus möglich, eine Built-in Function als Parameter in einer weiteren Built-in Function einzusetzen. Doch wo endet diese integrierte und kombinierte Verwendung der „Built-in Functions in Built-in Functions“ – also die Verschachtelung von Anweisungen?

Ein guter Programmierstil zeichnet sich durch wenige solcher Verschachtelungen aus.

Unsere Codierkollegen haben sich bereits vor geraumer Zeit mit derselben Thematik beschäftigt – und auch dort werden in der Praxis Verschachtelungen auf ein absolutes Minimum beschränkt.

Ein guter Ansatz für das Erstellen bzw. Anpassen von Programmen ist die Verwendung von nur einer einzigen Anweisung (bzw. nur einer Built-in Function) innerhalb einer Code-Zeile. Denn damit wird die Lesbarkeit des Programmcodes einfach gehalten. Auch wenn für die Definition und Zuordnung von Variablen vielleicht mehr Aufwand betrieben werden muss, so zahlt sich diese „Technik“ doch aus.

Ein gegensätzlicher Lösungsansatz ist die Maximierung der Verschachtelung der Anweisungen – also auch das Ziel, möglichst viele Code-Elemente (Anweisungen, Built-in Functions) in einer einzelnen Code-Zeile abbilden zu können. Mit diesem Lösungsansatz werden die Anzahl der im Programm erforderlichen Variablen und auch die Gesamtanzahl der Code-Zeilen reduziert. Dieses Ziel wird dann allerdings mit Hilfe von sehr komplexen und schwer zu lesenden Programmcodezeilen erreicht.

Wohl ist es der Mittelweg zwischen diesen beiden Techniken, der sich als „guter Programmierstil“ angeeignet werden sollte. Das bedeutet, dass Anweisungen oder auch Built-in Functions ineinander verschachtelt werden können. Diese Verschachtelungen sollten allerdings auf ein gesundes Maß begrenzt sein. Zwei, maximal drei Verschachtelungsstufen halte ich persönlich noch für praktisch einsetzbar, ohne dass die Lesbarkeit und Wartbarkeit des Codes darunter leidet.

Nachfolgende Abbildung enthält einige Beispiele für solche Verschachtelungen.

```
dcl-s char_a   char(8)  inz('28.02.18');
dcl-s char_b   char(10) inz ('13.01.2018');
dcl-s char_c   char(8) inz ('11.02.18');
dcl-s date_a   date    inz (D'2018-02-28');

dcl-s Num_A    zoned(6:0)  inz(280218);

num_A = %dec(%char(%date(char_A:*ymd.):*ymd0):6:0);

dsply num_A;


*inlr = *on;
```

Beispiel für eine dreifache Verschachtelung von Built-in Functions

3.2.8.13 Datums- und Zeitoperanten

Mit der Einführung der verschiedenen Datentypen, die im Zusammenhang mit Datums- und Zeitfunktionen stehen (solche Datentypen sind zum Beispiel „Date“, „Time“, „Timestamp“), entstanden auch einige neue Funktionen in RPG IV, mit denen diese neuen Datentypen verarbeitet werden können. So wurden erstmals in RPG IV die Operanten „Adddur“ (Hinzufügen einer Dauer) und „Subdur“ (Abziehen einer Dauer) realisiert.

Diese Operanten stehen im Free-Format-RPG nicht zur Verfügung. Stattdessen können für die Verarbeitung der Datums- und Zeitfelder neue Built-in Functions eingesetzt werden.

Konvertieren von numerischen Feldern in Datums- und Zeitformate
Für die Umsetzung von Daten, die in einem zeichen- oder numerischen Format vorliegen, in ein Datumsdatenformat, kann die Built-in Function %Date verwendet werden.

Der erste Parameter, der mit dieser Funktion eingesetzt wird, beinhaltet den umzusetzenden Wert – also ein Zeichenfeld oder ein numerisches Feld. Sollte der Wert des ersten Parameters ungültig sein, dann wird bei der Ausführung der Built-in Function eine Ausnahmemeldung ausgegeben. Mit dem zweiten Parameter wird das Format des ersten Eingabeparameters definiert. Sonderwerte, die hier als Parameter zum Einsatz kommen, beginnen alle mit dem Asterix-Zeichen (*). Die Verwendung des Formatparameters ist optional. Wird keine Angabe gemacht, dann wird der Unterlassungswert *ISO oder das Datumsformat verwendet, das in den H-Spezifikationen angegeben wurde. Als Rückgabewert gibt die Built-in Function %Date das Datum in einem Datumsformat zurück. Diese Funktion eignet sich auch für das Abrufen des Systemdatums. Dazu ist die Built-in Function %Date ohne weitere Parameter anzugeben.

Für die Umwandlung von Zeichen- oder numerischen Feldern in ein Uhrzeitformat kann die Built-in Function %Time verwendet werden. Der erste Parameter, der mit dieser Funktion angegeben werden kann, definiert den Eingabewert – also das Zeichenfeld oder das numerische Feld, das ins Zeitformat umgesetzt werden soll. Enthält dieser Parameter einen ungültigen Wert, dann wird eine Ausnahmemeldung ausgegeben. Der zweite Parameter gibt das Format an, in das der Wert des ersten Parameters umgesetzt werden soll. Hier stehen, wie auch bei der Built-in Function %Date, einige wenige Auswahlmöglichkeiten zur Verfügung. Sonderwerte beginnen immer mit dem „*“-Zeichen. Hier gilt, wie auch bei %Date, dass die Verwendung des zweiten Parameters optional ist. Wird dieser nicht angegeben, dann wird entweder *ISO als Format verwendet oder das Zeitformat, das in der H-Spezifikation definiert werden kann.

Die Built-in Function %Time kann auch dazu genutzt werden, um die aktuelle Systemzeit abzurufen. Dazu muss lediglich %Time – ohne weitere Parameter – angegeben werden.

Obwohl sie nur gelegentlich zum Einsatz kommt, so soll doch auch die Built-in Function %Timestamp für das Konvertieren eines Zeichenfeldes oder eines numerischen Feldes in ein Feld mit dem Format „Timestamp“ genutzt werden. Um den aktuellen Wert von „Timestamp“ abzurufen, geben Sie lediglich die Funktion %Timestamp ohne weitere Parameter an.

Konvertieren von Datums- oder Zeitfeldern in Zeichenfelder oder Dezimalfelder

Es gibt zwei Built-in Functions, mit denen die Umsetzung von Datums- und Zeitfeldern vorgenommen werden kann. Diese wurden bereits vorgestellt. Die Built-in Function %Char kann genutzt werden, um den Inhalt eines Datums- oder Zeitfeldes in ein Zeichenfeld umzuwandeln. In diesem Fall wird als erster Parameter das umzusetzende Datum bzw. die Zeit angegeben. Und mit dem zweiten Parameter wird das Ausgabeformat festgelegt. Hier existiert eine Reihe von unterschiedlichen Werten, die verwendet werden können. Die Ausgabe des umgewandelten Wertes erfolgt mit Datums- oder Zeitseparatoren, es sei denn, Sie geben direkt hinter dem Format „0“ an.

Seit der Version V5R3 kann die Built-in Function %Dec dazu verwendet werden, Datums- oder Zeitfelder in Felder mit Dezimalinhalt umzuwandeln. Der erste Parameter dieser Funktion beinhaltet das Datum bzw. die Zeit. Und der zweite Parameter legt das gewünschte Format fest, das ausgegeben werden soll. Hier existiert wieder eine bestimmte Vorgabeliste möglicher Formate. Sonderwerte beginnen mit dem „*“-Zeichen.

3.2.8
Seite 22

Nachfolgende Abbildung zeigt einige Beispiele für solche Datums- und Zeitumsetzungen.

```
dcl-s Datum8 Zoned(8:0)  inz(28022018);
dcl-s Datum1 Zoned(8:0)  inz(20180228);
dcl-s Datum2 Zoned(8:0)  inz;
dcl-s dDatum  date(*iso);
dcl-s Zeit6  zoned(6:0) inz (141116);
dcl-s ErgDat   date;
dcl-s ErgZeit  time;

dDatum = %date(Datum8:*eur);
dsply dDatum;

Datum2 =
    %dec(%char(%date(Datum1:*iso):
    *eur0):8:0);
dsply Datum2;

*inlr = *on;
```

Beispiele für die Umsetzung von Datums- und Zeitfeldern

3.2.8.14 Rechnen mit Datum und Zeit

Um mit Datums- und Zeitformaten arbeiten bzw. rechnen zu können, bieten sich im Free-Format-RPG IV der Einsatz von Datums- und Zeitvariablen und spezielle Built-in Functions an. Für die Verarbeitung von Datumsfeldern können die Built-in Functions %Days, %Month und %Years verwendet werden. Zeitfelder können mit den Funktionen %Hours, %Minutes, %Seconds und %Mseconds verarbeitet werden.

Jede dieser zuvor genannten Built-in Functions benötigt genau einen Parameter vom Typ Integer.

Free-Format-RPG IV bietet auch die Möglichkeit des Errechnens der Differenz zwischen zwei Datums- oder Zeitfeldern. Solche Rechenoperationen lassen sich mit Hilfe der Built-in Function %Diff für alle Felder mit den Typen „Date“, „Time“ oder „Timestamp“ durchführen.

Die Built-in Function %Subdt kann für das Abrufen eines vorgegebenen Bereichs eines Datums-, eines Zeit- oder eines Timestamp-Feldes verwendet werden. Der erste Parameter, der für die Funktion angegeben werden muss, ist das Feld, das umgesetzt bzw. dessen Inhalt teilweise abgerufen werden soll. Und der zweite Parameter definiert den Teilbereich, der extrahiert werden soll. Gültige Werte sind hier *Months, *Days und *Years für Datumsfelder sowie *Hours, *Minutes und *Seconds für Felder im Zeitformat. Alle vorhergehend genannten Werte und der Wert *Mseconds können auch in Verbindung mit einem Timestamp-Feld eingesetzt werden.

3.2.8
Seite 24

In folgender Abbildung finden sich Beispiele für die verschiedenen arithmetischen Built-in Functions in Verbindung mit Datums- und Zeitfeldern.

```
dcl-s Datum8 Zoned(8:0)  inz(28022018);
dcl-s Datum1 Zoned(8:0)  inz(20180228);
dcl-s Datum2 Zoned(8:0)  inz;
dcl-s dDatum  date(*iso);
dcl-s Zeit6  zoned(6:0) inz (141116);
dcl-s ErgDat   date;
dcl-s ErgZeit  time;

dDatum = %date(Datum8:*eur);
dsply dDatum;

Datum2 =
    %dec(%char(%date(Datum1:*iso):
    *eur0):8:0);
dsply Datum2;

*inlr = *on;
```

Built-in Functions für Datumsberechnungen

3.3 COBOL

Ergänzung 29./2014

3.3
Seite 2

3.3.1 CL – Control Language (Steuersprache)

CL ist ein Name, der verpflichtet – denn: Mit dieser Sprache soll(en) ihre Anwendung(en) gesteuert werden.

Allein wenn man an die unglaubliche Befehlsanzahl denkt (es sind aktuell immerhin zirka 1.900 Stück), wird einem schnell klar, um was für eine Sprache mit einem mächtigen, ja kaum überschaubaren Befehlsschatz es sich handelt.

Im Gegensatz dazu kennt RPG aktuell zirka 110 Op-Codes und 70 Built-in-Funktionen (BIFs) – und die sind ja zum Großteil nur Ersatz für OP-Codes.

Allerdings hinkt dieser Vergleich, denn CL dient ja „nur“ zur Anwendungssteuerung – und somit reduziert sich der dafür benötige Befehlsschatz auf eine überschaubare Anzahl von etwa 150 Befehlen. Ich bin mir auch im Klaren darüber, dass diese Zahl relativ ist. Was ich damit aber ausdrücken möchte: Versuchen Sie auf gar keinen Fall, CL als Ersatz für eine HLL-Sprache zu gebrauchen und beschränken Sie sich beim Einsatz dieser Sprache auf ihre Kernaufgabe.

Daher möchte ich dieses Kapitel zunächst darauf beschränken, was in den letzten Releases an Erweiterungen hinzugekommen ist.

Allein die Tatsache, dass mit jedem neuen Release auch immer wieder neue CL-Befehle hinzukommen (und die haben es zum Teil „in sich“) zeigt, das der Hersteller permanent an Verbesserungen und Neuerungen arbeitet.

3.3.1 Seite 2

3.3.2 CL – Neuerungen V5R2

Aus den Neuerungen in diesem Release möchte ich lediglich das Thema Imagekatalog herausstellen, der allerdings nur entfernt etwas mit dem Titel des Buchs zu tun hat. Aber – ich kann damit auf jeden Fall Installationsroutinen von Software-Produkten (einschließlich neuer Versionen von i5/OS oder PTF-Installationen) automatisieren und somit auch modernisieren.

Die Funktionen des Imagekatalogs erlauben es, Installations-CDs/DVDs in beliebiger Anzahl zunächst in ein i5-Objekt zu laden, um danach die eigentliche Installationsroutine aus einem virtuellen optischen Laufwerk auszuführen, das dem Imagekatalog zugeordnet wird. Das bedeutet, dass während der Installationsprozedur keine CD/DVD-Wechselaktivitäten mehr erforderlich sind. Ferner werden CD/DVD-Lesefehler, falsche Reihenfolgen usw. bei der eigentlichen Installationsprozedur dadurch ausgeschlossen, dass die Datenträger im Vorfeld bereits in den Imagekatalog geladen wurden. Solche Fehler treten dann bereits beim Ladevorgang auf – und es kann somit schon frühzeitig reagiert werden.

Die Imagekatalog-Befehle im Überblick:

Imagekatalogeintrag hinzufügen	ADDIMGCLGE
Imagekatalog ändern	CHGIMGCLG
Imagekatalogeintrag ändern	CHGIMGCLGE
Imagekatalog erstellen	CRTIMGCLG
Imagekatalog löschen	DLTIMGCLG
Imagekatalog laden/entladen	LODIMGCLG
Imagekatalogeintrag entfernen	RMVIMGCLGE
Imagekatalog prüfen	VFYIMGCLG
Mit Katalogeinträgen arbeiten	WRKIMGCLGE

Bild 1

Erwähnen möchte ich hier auch gleich folgenden Befehl:

Imagekatalog abrufen	RTVIMGCLG

Bild 2

Allerdings ist dieser Befehl erst seit V5R3 verfügbar.

3.3.2
Seite 2

Wichtig ist es zu wissen, dass immer nur ein Imagekatalog (LODIMGCLG) aktiv sein kann, dagegen ist die Anzahl der gespeicherten Imagekataloge praktisch unbegrenzt.

Anhand folgender Annahme soll die Verfahrensweise erläutert werden:

- Sie erhalten sechs CDs mit einem kumulativen PTF.
- Der eigentliche, echte Installationsvorgang soll ohne weitere Eingriffe (eventuell in den Nachtstunden) ausgeführt werden.
- Sie benötigen auf jeden Fall ein virtuelles, optisches Laufwerk. Dies wird (einmalig) erstellt mit:

```
      Einheitenbeschreibung erstellen (Opt.) (CRTDEVOPT)

Auswahl eingeben und Eingabetaste drücken.

Einheitenbeschreibung  . . . . . DEVD          (1) nnnnnn
Ressourcenname . . . . . . . . . RSRCNAME      (2) *vrt
Einheitentyp . . . . . . . . . . TYPE              *RSRCNAME
Bei IPL anhängen . . . . . . . . ONLINE            *YES
Nachrichtenwarteschlange . . . . MSGQ              *SYSOPR
  Bibliothek . . . . . . . . . .
Text 'Beschreibung'  . . . . . . TEXT          (3) virtuelles
                                                    CD-Laufwerk
```

Bild 3

1 = Name des virtuellen Laufwerks

2 = Hier muss *VRT (virtuell) angegeben werden

3 = Beschreibung (Dokumentation)

Erstellen eines Imagekatalogs:

```
              Imagekatalog erstellen (CRTIMGCLG)
  Auswahl eingeben und Eingabetaste drücken.

  Imagekatalog . . . . . . . . . . IMGCLG    4 nnnnnn
  Verzeichnis  . . . . . . . . . . DIR       5 /nnnnn

                    Zusätzliche Parameter
  Verzeichnis erstellen  . . . . . CRTDIR    6 *YES

  Text 'Beschreibung'  . . . . . . TEXT      7 nnnnnn

  Berechtigung . . . . . . . . . . AUT           *EXCLUDE
```

Bild 4

4 = Name des Imagekatalogs

5 = Verzeichnisname, der dem Imagekatalog zugeordnet wird

6 = Verzeichnis wird angelegt, falls es noch nicht existiert

7 = Beschreibung (Dokumentation)

Laden (Schreiben) der CDs/DVDs in den Imagekatalog:

```
         Imagekatalogeintrag hinzufügen (ADDIMGCLGE)
  Auswahl eingeben und Eingabetaste drücken.
  Imagekatalog . . . . . . . . . . IMGCLG         nnnnnn
  Von optischer Einheit oder . . . FROMDEV    8 opt01
  Aus Imagedatei . . . . . . . . . FROMFILE   9

  In Imagedatei  . . . . . . . . . TOFILE         *GEN

  Imagekatalogindex  . . . . . . . IMGCLGIDX      *AVAIL
 Laden der CDs/DVDs als Eintrag in den Imagekatalog:
```

Bild 5

8 = Name des Imagekataloges, in den die CD/DVD geladen werden soll (Name aus 4)

9 = Name des physikalischen CD-Laufwerks (in der Regel = OPT01)

3.3.2
Seite 4

Dieser Befehl ist für jede CD/DVD zu wiederholen.

Zuordnen (Aktivieren) des Imagekatalogs zum virtuellen optischen Laufwerk:

```
          Imagekatalog laden/entladen (LODIMGCLG)
Auswahl eingeben und Eingabetaste drücken.

Imagekatalog . . . . . . . . . . IMGCLG     10 nnnnnn
Virtuelle optische Einheit . . . DEV        11 nnnnnn
Auswahl  . . . . . . . . . . . . OPTION     12 *LOAD
```

Bild 6

10 = Name des Imagekatalogs (Name aus 4 bzw. 8)

11 = Name der virtuellen, optischen Einheit (Name aus 1)

12 = Katalog soll geladen werden – in das virtuelle, optische Laufwerk (im Gegensatz zum UNLOAD = trennen)

Eventuell Imagekatalog prüfen – u.a. auf Vollständigkeit, korrekte Reihenfolge etc. Allerdings ist das kein Muss, man sollte aber diese Aktion durchführen, um sicher zu gehen, dass alles in Ordnung ist:

```
               Imagekatalog prüfen (VFYIMGCLG)

Auswahl eingeben und Eingabetaste drücken.

Imagekatalog . . . . . . . . . . IMGCLG       nnnnnn
Imagekatalogtyp  . . . . . . . . TYPE       13 *ptf
Imagekatalog sortieren . . . . . SORT       14 *NO
```

Bild 7

13 = Name des Imagekatalogs (Name aus 4, 8 und 10)

14 = Inhalt des Imagekatalogs ist vom Typ PTF

Eigentliches PTF laden:

```
                PTF installieren (INSPTF)

Auswahl eingeben und Eingabetaste drücken.

Produktbeschreibung:             LICPGM
  Produkt  . . . . . . . . . . .          15 *all
  Release  . . . . . . . . . . .             *ONLY
                       + für weitere Werte
Einheit  . . . . . . . . . . . . DEV       16 nnnnnn
Art des Anlegens . . . . . . . . INSTYP       *SRVATT
PTF-Ausschlussliste:             OMIT
  Produkt  . . . . . . . . . . .
  PTF-ID . . . . . . . . . . . .
  Release  . . . . . . . . . . .             *ONLY
                       + für weitere Werte
```

Bild 8

15 = *ALL: alle PTFs sind zu laden

16 = Name der virtuellen und optischen Einheit

3.3.2
Seite 6

Mit dem Imagekatalog kann auch die Installation eines neuen i5/OS Release durchgeführt werden. Das Laden der CDs/DVDs, das Laden des Imagekatalogs in das virtuelle Laufwerk und das eventuell vorherige Prüfen laufen genau nach dem oben beschriebenen Schema ab.

Anstelle des INSPTF wäre hier dann folgender Befehl erforderlich:

```
                System ausschalten (PWRDWNSYS)

Auswahl eingeben und Eingabetaste drücken.

Beenden  . . . . . . . . . . . . . OPTION         *CNTRLD
Verzögerungszeit, wenn *CNTRLD    DELAY          3600
Angaben für Wiederanlauf:         RESTART
  Neustart nach Abschalten . . .                 *NO
  Art des Wiederanlaufs  . . . .                 *IPLA
IPL-Quelle . . . . . . . . . . . . IPLSRC       > *IMGCLG  17
Imagekatalog . . . . . . . . . . . IMGCLG         nnnnnn   18
```

Bild 9

17 = Die IPL-Quelle ist ein Imagekatalog

18 = Name des Imagekatalogs (Name aus 4, 8 und 10)

Damit kann die komplette Installationsprozedur ohne weitere manuelle Eingriffe laufen.

3.3.3 CL – Neuerungen V5R3

3.3.3.1 Neue Typen von Variablen

Außer den bereits bekannten Typen

*CHAR,

*DEC und

*LGL

kann man in CL jetzt auch die Typen

*INT und

*UINT

definieren.

*INT ist eine ganzzahlige Variable, die einen Binärwert mit Vorzeichen enthält. Die maximale und gleichzeitige Standardlänge beträgt 4 Byte. Gültige Längenangaben sind 2 oder 4 Byte. Der Wertebereich (Inhalt) für diesen Feldtyp liegt zwischen:

-32.768	und	+ 32.767	bei 2 Byte Länge
-2.147.483.648	und	+2.147.483.646	bei 4 Byte Länge

*UINT ist ebenfalls eine ganzzahlige Variable die einen Binärwert – allerdings ohne Vorzeichen – enthält. Die maximale und auch gleichzeitige Standardlänge beträgt 4 Byte. Gültige Längenangaben sind 2 oder 4 Byte. Der Wertebereich (Inhalt) für diesen Feldtyp liegt zwischen:

0	und	65.535
0	und	4.294.967.295

3.3.3.2 Deklaration und Verarbeitung von Dateien

Die maximale Anzahl von Dateien (=DCLF-Anweisungen) wurde auf fünf erhöht.

Wird mehr als eine Datei in einem CL-Programm definiert, ist es unabdingbar, dem neuen Schlüsselwort OPNID bei der DCLF-Anweisung einen Parameterwert für jede Datei zuzuordnen. Dieser Wert muss eindeutig sein. Somit ist es

3.3.3
Seite 2

auch möglich, eine Datei gleichen Namens mehrfach in einem CL-Programm zu definieren.

Die DCLF-Anweisung:

```
                  Datei deklarieren (DCLF)

Auswahl eingeben und Eingabetaste drücken.

                                                        Dateiname
Datei  . . . . . . . . . . . . . . FILE           file01
  Bibliothek . . . . . . . . . .                    *LIBL
Satzformat . . . . . . . . . . . . RCDFMT         *ALL
               + für weitere Werte
                                                        Datei-ID
ID der offenen Datei . . . . . . . OPNID          f01
Felder variabler Länge zulassen  ALWVARLEN        *NO
Null als Feldwert zulassen . . . ALWNULL          *NO
Grafikfelder zulassen  . . . . . ALWGRAPHIC       *NO
```

Bild 10

Auch die CL-Anweisungen zum Verarbeiten von Dateien

```
SNDF/SNDRCVF/SNDRCVF/RCVF
```

wurden um das Schlüsselwort OPNID erweitert.

Das bedeutet, dass man die Dateien nicht mehr direkt über den Dateinamen aus der DCLF-Anweisungen verarbeitet, sondern über die angegebene ID (den Namen) von OPNID. Vergleichbar ist das mit dem Cursor bei SQL.

Außerdem werden alle Felder der jeweiligen Datei, die ja automatisch zu CL-Variablen werden, um die bei OPNID angegebene ID als Präfix ergänzt – gefolgt von einem Unterstrich („_“) und dem eigentlichen Feldnamen (in obigem Beispiel um F01_).

3.3.3.3 Strukturbefehle im CL

Wenn im CL-Programm eine Schleife benötigt (z.B. beim Lesen von Dateien) wird, war man bisher gezwungen, diese mit GOTO zu programmieren.

Dies ist nun hinfällig durch die neuen CL-Strukturbefehle:

- DOWHILE
- DOUNTIL
- DOFOR
- ITERATE
- LEAVE
- SELECT
- WHEN
- OTHERWISE
- ENDSELECT

DOWHILE und DOUNTIL sind absolut identisch mit dem Äquivalent im RPG.

```
                Ausführen während (DOWHILE)


Auswahl eingeben und Eingabetaste drücken.
                                          Angabe der
                                          Schleifenbedingung
Bedingung  . . . . . . . . . . . COND
```

Bild 11

```
                  Ausführen bis (DOUNTIL)


Auswahl eingeben und Eingabetaste drücken.
                                          Angabe der
                                          Schleifenbedingung
Bedingung  . . . . . . . . . . . COND
```

Bild 12

3.3.3
Seite 4

DOFOR ist geringfügig anders. In RPG wird bei FOR das Schlüsselwort DOWNTO angegeben, sofern der Schleifenerhöhungswert subtrahiert werden soll. Bei der DOFOR-Anweisung in CL muss man zu diesem Zweck das Schlüsselwort BY mit einem negativen Parameterwert angeben.

```
                    Ausführen für (DOFOR)

Auswahl eingeben und Eingabetaste drücken.

Name der CL-Variablen  . . . . . VAR   (1)
Eingabewert  . . . . . . . . . . FROM  (2)
Bis Wert . . . . . . . . . . . . TO    (3)
Nach Wert  . . . . . . . . . . . BY    (4)       1
```

Bild 13

1 Name der CL-Variablen, die zur Steuerung der Schleife verwendet wird. Die Variable muss als Art *INT oder *UINT definiert werden.

2 Definiert den Anfangswert der Schleife. Die Variable muss als Art *INT oder *UINT definiert werden.

3 Definiert den Endwert der Schleife. Die Variable muss als Art *INT oder *UINT definiert werden.

4 Gibt den Wert an, der zur Steuerungsvariablen (1) nach jedem Schleifendurchlauf hinzugefügt (addiert oder subtrahiert) wird. Der Wert muss als ganzzahlige Konstante angegeben werden (Default = 1) und kann null, negativ oder positiv sein.

Die Werte der Parameter VAR (1) und TO (3) können während der Schleifendurchläufe verändert werden. Dann können eben mehr oder weniger Schleifendurchläufe durchgeführt werden, als dass zu Beginn der Schleife angegeben war.

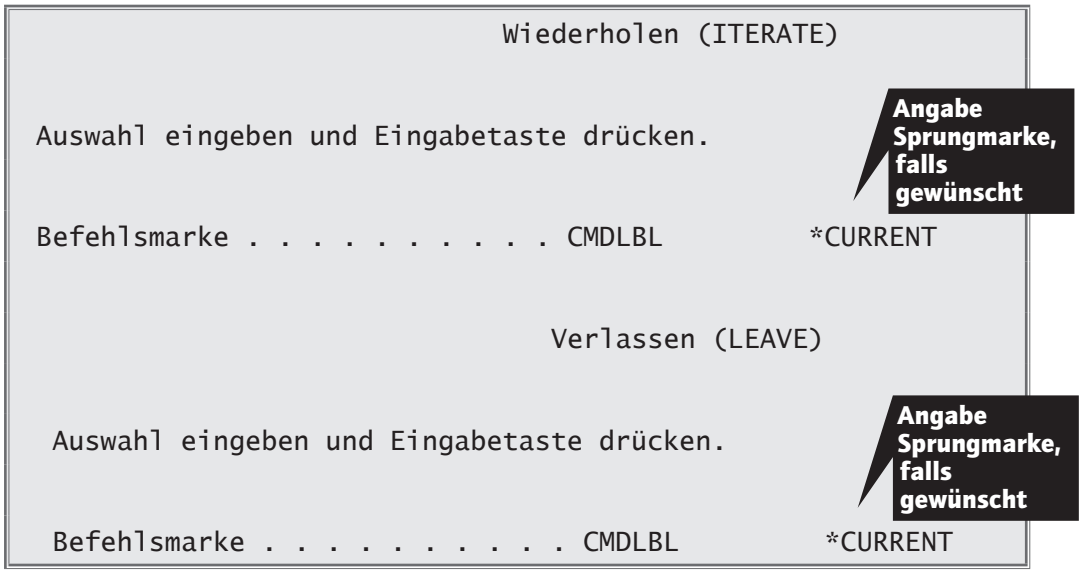


Bild 14

ITERATE und LEAVE weisen auch eine Besonderheit auf, die es so im Äquivalent des RPG nicht gibt. Wenn der Parameter beim Schlüsselwort *CURRENT bleibt, arbeiten beide Befehle genauso wie im RPG. ITERATE verzweigt auf die ENDDO-Anweisung der Schleife, innerhalb der die Anweisung ITERATE angegeben ist. LEAVE verzweigt auf die nächste Anweisung, die dem ENDDO der Schleife folgt.

Bei Befehlsmarke bzw. beim Schlüsselwort CMDLBL kann eine Sprungmarke (= TAG) angegeben werden. Aber keine Bange, diese Marke ist kein wahlfreier TAG irgendwo im CL-Programm, sondern er muss bei einer Schleifenanweisung (DOWHILE, DOUNTIL, DOFOR) angegeben worden sein. Diese muss wiederum in einer sogenannten aktiven Gruppe vorhanden sein.

3.3.3
Seite 6

```
MARKE_1:    DOWHILE    COND(&VAR1 <= 20)

            ENDDO

MARKE_2:    DOWHILE    COND(&VAR2 > 100)

               DOUNTIL    COND(&VAR3 = 'ABC')



IF          COND(&VAR_47111 = 'BANANE') THEN(LEAVE +
                           (1) CMDLBL(MARKE1))

IF          COND(&VAR_47111 = 'BANANE') THEN(LEAVE +
                           (2) CMDLBL(MARKE2))

               ENDDO

            ENDDO
```

Bild 15

1 Nicht erlaubt, da Marke nicht innerhalb der aktiven Gruppe ist (Verschachtelung)

2 Erlaubt

Dies ist eine elegante Erweiterung gegenüber dem Äquivalent im RPG.

Auch die SELECT-Gruppe ist mit den entsprechenden Anweisungen des RPG vergleichbar, wenngleich es auch hier wieder einen Unterschied gibt.

In RPG gilt eine Abfrage vom Beginn einer WHEN-, OTHER- oder ENDCS-Anweisung bis zur folgenden WHEN-, OTHER- oder ENDCS-Anweisung.

Ähnlich wie beim IF kann bei einer WHEN-Anweisung beim Schlüsselwort THEN nur ein CL-Befehl angegeben werden. Sollen auf Grund einer WHEN-Bedingung mehrere CL-Befehle ausgeführt werden so bietet sich an, bei Befehl (Schlüsselwort THEN) die Anweisung DO anzugeben. Innerhalb der DO-Gruppe können dann mehrere CL-Befehle eingebettet werden (Abschluss durch ENDDO).

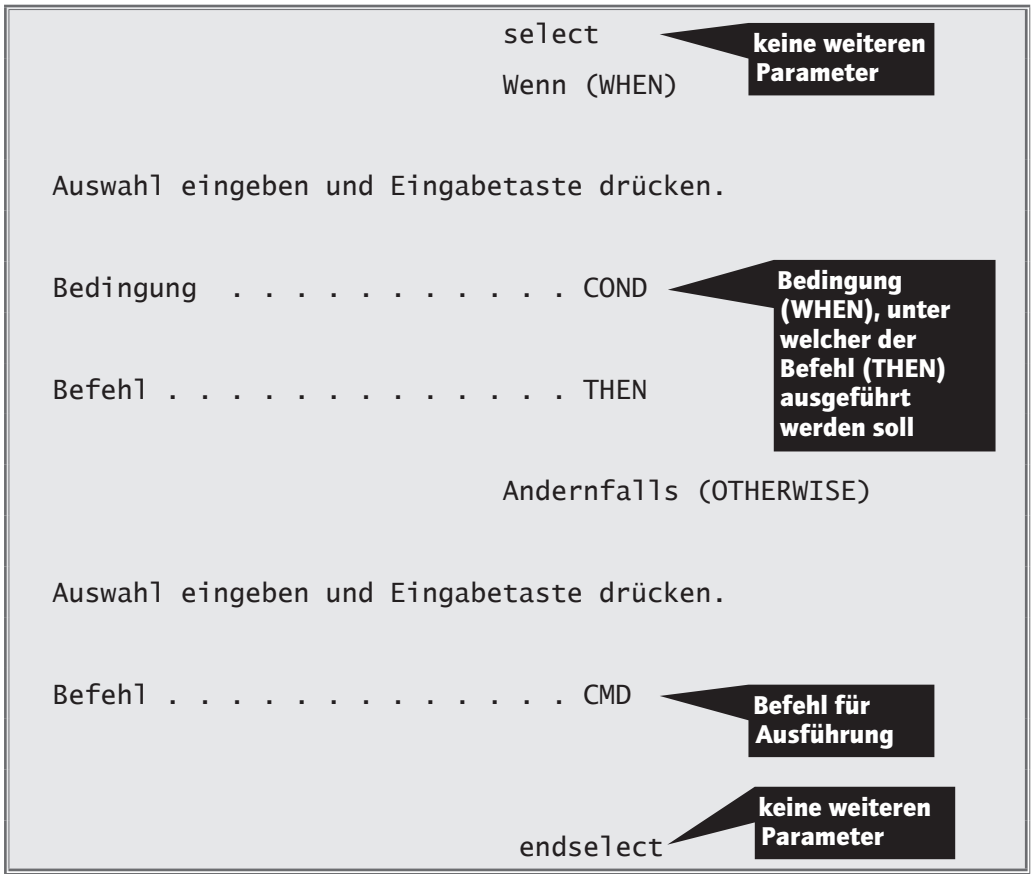

select
Wenn (WHEN)
keine weiteren Parameter
Auswahl eingeben und Eingabetaste drücken.
Bedingung . . . . . COND
Befehl . . . . . THEN
Bedingung (WHEN), unter welcher der Befehl (THEN) ausgeführt werden soll
Andernfalls (OTHERWISE)
Auswahl eingeben und Eingabetaste drücken.
Befehl . . . . . CMD
Befehl für Ausführung
endselect
keine weiteren Parameter

Bild 16

3.3.3
Seite 8

3.3.3.4 Die neuen CL-Befehle in einem Musterprogramm

Zur praktischen Illustration verschiedener, neuer CL-Befehle erfolgt nun ein kleines Beispielprogramm. In diesem Programm sollen zwei Dateien verarbeitet werden, die durch ein Vorprogramm erstellt wurden. Die erste Datei ist ein Produkt aus folgendem CL-Befehl:

```
                    Dateibeschreibung anzeigen (DSPFD)

Auswahl eingeben und Eingabetaste drücken.

Datei  . . . . . . . . . . . . . > *ALL          Name, generisch*, *ALL
  Bibliothek . . . . . . . . . . >   ITP         Name, *LIBL, *CURLIB...
Informationsart  . . . . . . . . > *ATR          *ALL, *BASATR, *ATR...
     + für weitere Werte
Ausgabe  . . . . . . . . . . . . > *OUTFILE      *, *PRINT, *OUTFILE
Dateiattribute . . . . . . . . . > *PF           *ALL, *DSPF, *PRTF, …
     + für weitere Werte
Datei für Ausgabe  . . . . . . . > ##DSPFD       Name
  Bibliothek . . . . . . . . . .     *LIBL       Name, *LIBL, *CURLIB
Auswahl für Ausgabeteildatei:
  Teildatei für Ausgabe  . . . .   *FIRST        Name, *FIRST
  Sätze ersetzen od. hinzufügen    *REPLACE      *REPLACE, *ADD
```

Bild 17

Die zweite Datei resultiert aus folgendem CL-Befehl: **3.3.3**

```
                    Dateibeschreibung anzeigen (DSPFD)

Auswahl eingeben und Eingabetaste drücken.

Datei  . . . . . . . . . . . . . . > *ALL          Name, generisch*, *ALL
  Bibliothek . . . . . . . . . . . >   ITP         Name, *LIBL, *CURLIB...
Informationsart  . . . . . . . . . > *MBR          *ALL, *BASATR, *ATR...
     + für weitere Werte
Ausgabe  . . . . . . . . . . . . . > *OUTFILE      *, *PRINT, *OUTFILE
Dateiattribute . . . . . . . . . . > *PF           *ALL, *DSPF, *PRTF, ...
     + für weitere Werte
Datei für Ausgabe  . . . . . . . . > ##MEMBR       Name
  Bibliothek . . . . . . . . . . .     *LIBL       Name, *LIBL, *CURLIB
Auswahl für Ausgabeteildatei:
  Teildatei für Ausgabe  . . . .   *FIRST         Name, *FIRST
  Sätze ersetzen od. hinzufügen    *REPLACE       *REPLACE, *ADD
```

Bild 18

3.3.3
Seite 10

```
            PGM

            DCL         VAR(&NBRCURRCD) TYPE(*DEC) LEN(10)
            DCL         VAR(&VAR_F01  ) TYPE(*UINT)
            DCL         VAR(&TO_F01   ) TYPE(*UINT)
            DCL         VAR(&EOF_F02) TYPE(*LGL)
            DCL         VAR(&TRUE) TYPE(*LGL) VALUE('1')
            DCL         VAR(&FALSE) TYPE(*LGL) VALUE('0')

            DCLF        FILE(QAFDPHY) OPNID(F01)
            DCLF        FILE(QAFDMBR) OPNID(F02)

            OVRDBF      FILE(QAFDPHY) TOFILE(##DSPFD)

            RTVMBRD     FILE(QTEMP/##DSPFD) NBRCURRCD(&NBRCURRCD)
            CHGVAR      VAR(&TO_F01) VALUE(&NBRCURRCD)

            OVRDBF      FILE(QAFDPHY) TOFILE(##DSPFD)

            DOFOR       VAR(&VAR_F01) FROM(1) TO(&TO_F01) BY(1)
               RCVF        OPNID(F01)

SNDMSG      MSG('Verarbeiten der Datei' *BCAT &F01_PHLIB +
                        *TCAT '/' *TCAT &F01_PHFILE *BCAT 'wird +
                        ausgeführt') TOMSGQ(ITP_WB/REORG_MSG)

   /*VERARBEITEN DER DATEN AUS DATEI ##DSPFD                          */

            ENDDO

            DLTOVR      FILE(QAFDPHY)
            MONMSG      MSGID(CPF9841)
```

Ermitteln Anzahl Sätze in Datei für DOFOR

Verwendung einer DOFOR-Schleife als Beispiel (könnte auch DOWHILE/DOUNTIL sein)

Hier würde dann die eigentliche Verarbeitung stattfinden

Bild 19_1

```
OVRDBF      FILE(QAFDMBR) TOFILE(##MEMBR)


DOWHILE     COND(&EOF_F02 = &FALSE)
   RCVF         OPNID(F02)
   MONMSG       MSGID(CPF0864) EXEC(LEAVE)

      IF          COND(&F02_MBNAME = ' ') THEN(ITERATE)

      SNDMSG      MSG('Verarbeiten der Teildatei' *BCAT +
                    &F02_MBNAME *BCAT 'wird ausgeführt') +
                    TOMSGQ(ITP_WB/REORG_MSG)

   SELECT

      WHEN        COND(&F02_MBACCP = 'K') +
          THEN(RGZPFM   FILE(&F01_PHLIB/&F01_PHFILE) +
              MBR(&F02_MBNAME) KEYFILE(*FILE))

     OTHERWISE  CMD(RGZPFM FILE(&F01_PHLIB/&F01_PHFILE) +
                    MBR(&F02_MBNAME))

      ENDSELECT

   ENDDO

   DLTOVR      FILE(QAFDMBRL)
   MONMSG      MSGID(CPF9841)

ENDPGM
```

Verwendung einer DOWHILE-Schleife

SELECT mit WHEN und OTHERWISE

Bild 19_2

3.3.3
Seite 12

3.3.3.5 Informationen aus der Umwandlungsliste.

```
     DCLF       FILE(QAFDPHY) OPNID(F01)
```

Definition der ersten Datei mit OPNID = F01

```
 QUALIFIZIERTER DATEINAME - QSYS/QAFDPHY
   SATZFORMATNAME - QWHFDPHY                                DSPFD-Format für TYPE(*ATR) FILEATR(*PF)
      CL-VARIABLE              ART      LÄNGE  GENAUIGKEIT  TEXT
       &F01_PHRCEN             *CHAR       1                Abrufjahrhundert:  0
       &F01_PHRDAT             *CHAR       6                Abrufdatum:
       &F01_PHRTIM             *CHAR       6                Abrufzeit:
       &F01_PHFILE             *CHAR      10                Datei
       &F01_PHLIB              *CHAR      10                Bibliothek
       &F01_PHFTYP             *CHAR       1                Dateiart: P=PF,
       &F01_PHFILA             *CHAR       4                Dateiattribut:  *PHY
```

Feldnamen mit OPNID als Prefix

```
   DCLF       FILE(QAFDMBR) OPNID(F02)
```

Definition der zweiten Datei mit der OPNID = F02

```
  QUALIFIZIERTER DATEINAME - QSYS/QAFDMBR
   SATZFORMATNAME - QWHFDMBR                                    DSPFD-Format für TYPE(*MBR)
CL-VARIABLE              ART      LÄNGE      GENAUIGKEIT   TEXT

&F02_MBRCEN             *CHAR       1                     Abrufjahrhundert:  0=19xx,
&F02_MBRDAT             *CHAR       6                     Abrufdatum: Jahr.Monat.Tag
&F02_MBRTIM             *CHAR       6                     Abrufzeit:
&F02_MBFILE             *CHAR      10                     Datei
&F02_MBLIB              *CHAR      10                     Bibliothek
&F02_MBFTYP             *CHAR       1                     P=PF, L=LF, R=DDM PF, S=DDM
&F02_MBFILA             *CHAR       4                     Dateiattribut: *PHY oder *LGL
&F02_MBMXD              *CHAR       3                     Reserviert
&F02_MBFATR             *CHAR       6                     Dateiattribut:  PF, LF, PF38
&F02_MBSYSN             *CHAR       8                     Systemname (Quellensystem, )
&F02_MBASP              *DEC        3           0         ID des Zusatzspeicher-Pools:
&F02_MBRES              *CHAR       4                     Reserviert
```

Feldnamen mit OPNID als Prefix

Bild 20

3.3.4 CL – Neuerungen V5R4

3.3.4.1 Erweiterungen in der DCL-Anweisung

Die Erweiterungen in der DCL-Anweisung:

```
                CL-Variable deklarieren (DCL)

Auswahl eingeben und Eingabetaste drücken.

Marke  . . . . . .
Name der CL-Variablen  . . . . . VAR
Art  . . . . . . . . . . . . . . TYPE
Speicher . . . . . . . . . . . . STG          *AUTO
Variablenlänge:                  LEN
  Länge  . . . . . .
  Dezimalstellen . . . . . .
Anfangswert  . . . . . . . . . . VALUE
 ...
Basiszeigervariable  . . . . . . BASPTR

In Variable definiert:           DEFVAR
  Name der CL-Variablen  . . . .
  Position . . . . . .
Adresse:                         ADDRESS
  Name der CL-Variablen  . . . .
  Abweichung . . . . . .
Kommentar  . . . . . .
```

1 2 3 4 5 6

Bild 21

1 Neuer Variablen-Typ der Art Pointer (*PTR)

2 Neues Schlüsselwort STG. Folgende Parameterwert sind erlaubt:
2.1. *AUTO = Definition einer einzelnen Variable (Feld). Dies ist der Standardwert.

2.2. *DEFINED = Definition eines Datenstruktursubfeldes

2.3 *BASED = Definition einer Based-Variablen

3 Die Länge von alphanumerischen Variablen (TYPE(*CHAR)) wurde auf 32.767 Byte erweitert.

4 Based Pointer Variable: Erforderlich, wenn STG(*BASED) angegeben wurde.

5 Datenstruktur-Unter(Sub)feld: Erforderlich, wenn STG(*DEFINED) angegeben wurde.

6 Initial-Adresse für Pointer-Variable: Kann bei TYP(*PTR) und STG(*AUTO) angegeben werden.

3.3.4.2 Definition von Datenstrukturen

Jetzt kommen wir zu einer Neuerung, die es in sich hat. Gerade der Aufruf von System-APIs gestaltete sich in der Vergangenheit von CL relativ problematisch. APIs arbeiten sehr oft mit Datenstrukturen, was aber in gleichem Maß auch für eigene Programme gilt. Man hat sich mit der Übergabe von alphanumerischen Variablen beholfen, was umfangreiche Substrings (%SUBTRS/%SST) notwendig machte – und zwar deshalb, um den String zu bilden bzw. um ihn nach Empfang in seine einzelnen Bestandteile zu zerpflücken.

Abhilfe schafft jetzt die Möglichkeit der Definition von Datenstrukturen. Man definiert eine Variable und – ähnlich wie im RPG – unterteilt man diese Variable in Unter-(Sub-) Felder.

Selbstverständlich ist es auch möglich, durch unterschiedliche Definitionen den gleichen Bereich mit unterschiedlichen Typen (z. B. einmal als *CHAR und einmal als *DEC) zu definieren bzw. den gleichen Bereich mehrfach, auch mit unterschiedlichen Längenangaben zu überlagern.

Folgendes Beispiel zeigt den sinnvollen Einsatz von Datenstrukturen im CL.

In einem Datenbereich (*DTAARA) stehen diverse Informationen in einer Gesamtlänge von 100 Byte. Diese werden eingelesen und müssen jetzt nach dem Einlesen nicht mehr mit %SST zerpflückt werden:

Definition von Datenstrukturen:

```
DCL        VAR(&DTAARA) TYPE(*CHAR) STG(*AUTO) LEN(100)   (1)




DCL        VAR(&FELD_01) TYPE(*CHAR) STG(*DEFINED) +   (2)
                         LEN(10) DEFVAR(&DTAARA)

DCL        VAR(&FELD_02) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(50) DEFVAR(&DTAARA 11)

DCL        VAR(&FELD_03) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(30) DEFVAR(&DTAARA 61)

DCL        VAR(&FELD_04_C) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(5) DEFVAR(&DTAARA 91)
DCL        VAR(&FELD_04_N) TYPE(*DEC) STG(*DEFINED) +
                         LEN(5 0) DEFVAR(&DTAARA 91)

DCL        VAR(&FELD_05_1) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(5) DEFVAR(&DTAARA 96)
DCL        VAR(&FELD_05_2) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(4) DEFVAR(&DTAARA 96)
DCL        VAR(&FELD_05_3) TYPE(*CHAR) STG(*DEFINED) +
                         LEN(3) DEFVAR(&DTAARA 98)

RTVDTAARA  DTAARA(BANANE (1 100)) RTNVAR(&DTAARA)   (3)
```

Bild 22

1 Definition der Datenstruktur. Diese wird zunächst wie eine Variable definiert.
Die maximale Länge sind 32.767 Byte. In RPG können Datenstrukturen bis zu einer Länge von 65.535 definiert werden. Das ist zu beachten, falls zwischen CL und RPG Datenstrukturen als Parameter ausgetauscht werden.

2 Ab hier beginnt die Definition der Datenstrukturunterfelder und die Re-Definition der Variablen &DTAARA. Wichtig ist die Angabe von STG(*DEFINED) und DEFVAR. Bei diesem Schlüsselwort wird angegeben, auf welche Variable sich die Re-Definition bezieht und ab welcher Stelle das Datenstrukturunterfeld (Subfeld) beginnt. Vergleichbar mit OVERLAY im RPG.

3 Einlesen des Datenbereiches für das angenommene Beispiel.

3.3.4.3 Pointer Variablen und Pointer-Arithmetik im CL

In der klassischen Programmierung mit CL, RPG oder COBOL ist die Benutzung von Pointern meistens nicht erforderlich, da man sich auf die Verwendung von Feldnamen beschränkt. Feldnamen sind aber nichts anderes als symbolische Speicheradressen – also Pointer.

In Pointern, die nun in CL definiert werden können, werden die Adressen von Variablen oder Datenstrukturen gespeichert.

Eine Pointer-Variable wird über die DCL-Anweisung definiert werden, wobei der TYP(*PTR) angegeben werden muss. Eine Längenangabe erfolgt nicht; eine Pointer-Variable wird immer in der Länge von 16 Byte definiert.

Eine Angabe von VALUE ist bei der DCL-Anweisung ebenfalls nicht erlaubt. Allerdings kann man bei der DCL-Anweisung beim Schlüsselwort ADRESS dem Pointer quasi als Initialwert die Adresse einer Variablen/Datenstruktur zuordnen. Als Option kann außerdem ein sogenannter OFFSET angegeben werden. Der OFFSET verweist auf die Stelle (Byte) innerhalb der Variablen/Datenstruktur. Ein OFFSET beginnt immer mit dem Wert 0 und meint damit die erste Stelle innerhalb der Variablen/Datenstruktur. So verweist beispielsweise eine OFFSET-Angabe 75 auf das Byte 76 innerhalb der Variablen/Datenstruktur.

```
                          CL-Variable deklarieren (DCL)

Auswahl eingeben und Eingabetaste drücken.

Marke  . . . . . .
Name der CL-Variablen  . . . . . VAR
Art  . . . . . . . . . . . . . . TYPE
Speicher . . . . . . . . . . . . STG             *AUTO
Variablenlänge:                  LEN
  Länge  . . . . . .
  Dezimalstellen . . . . . .
Anfangswert  . . . . . . . . . . VALUE
 ...
Basiszeigervariable  . . . . . . BASPTR

In Variable definiert:           DEFVAR
  Name der CL-Variablen  . . . .
  Position . . . . . .                    1
Adresse:                         ADDRESS
  Name der CL-Variablen  . . . .
  Abweichung . . . . . .                  0
Kommentar  . . . . . .
```

Type = *PTR

keine Längenangabe

Zuordnung der Adresse einer CL-Variablen als Initialwert

Bild 23

3.3.4
Seite 6

Nun würde es wenig Sinn machen, während eines Programmdurchlaufs immer mit der gleichen Adresse, die ja lediglich auf eine Variable/Datenstruktur verweist, zu arbeiten. Es muss die Möglichkeit geben, die Adresse quasi zu verschieben, entweder auf andere Positionen innerhalb der gleichen oder auf andere Variablen/Datenstruktur(en).

Dies erfolgt mit Built-in-Funktionen:

%Adress oder %Adr	um auf andere Variablen/Datenstrukturen zu verweisen.
%Offset oder %Ofs	um innerhalb der Variablen/Datenstruktur auf eine bestimmte Stelle zu verweisen.

```
                  Variable ändern (CHGVAR)

Auswahl eingeben und Eingabetaste drücken.

Marke  . . . . . .
Name der CL-Variablen  . . . . . > &POINTER
Neuer Wert . . . . . > %ADDRESS(&BANANE)
```

Bild 24

```
                  Variable ändern (CHGVAR)

Auswahl eingeben und Eingabetaste drücken.

Marke  . . . . . .
Name der CL-Variablen  . . . . . > %OFFSET(&POINTER)
Neuer Wert . . . . . > (%OFS(&POINTER) + 20)
```

Bild 25

%Adress oder %Adr kann auch zur Abfrage verwendet werden, ob einem Pointer bereits eine bestimmte Adresse zugeordnet wurde.

Am einfachsten kann man Pointer mit Feldgruppen erklären.

Eine Feldgruppe hat einen Namen und damit eine Adresse. Ein Pointer, der diesem Namen zugeordnet ist, verweist auf diese Adresse. Durch die Längenangabe eines Elements und die Anzahl der Elemente ergibt sich eine Gesamtlänge

– der OFFSET der Feldgruppe insgesamt. Sobald ein einzelnes Element der Feldgruppe angesprochen (Index) wird, wird die Adresse – der OFFSET – auf dieses Element verschoben.

Um Pointer sinnvoll einzusetzen, ist es notwendig, auf die Daten ab der Position zugreifen zu können, auf die der Pointer gesetzt wurde.

Das geschieht über „Based-Variablen".

Für Based-Variablen wird kein Speicherplatz reserviert, dafür werden sie einem Pointer zugeordnet.

Das Beispiel aus 4.2 soll dahingehend erweitert werden, dass im Datenbereich die 100 Byte, die dort über eine Datenstruktur in Unterfelder aufgeteilt wurden, insgesamt 10 mal vorkommen. Nun kann man zwar auch die Datenstruktur so erweitern, dass man alle 100 Byte über die Datenstruktur in Unterfelder aufteilt. Geschickter ist aber die Verwendung von Pointer und Based-Variablen.

Beispielprogramm:

```
DCL        VAR(&VAR      ) TYPE(*UINT)
DCL        VAR(&FROM) TYPE(*UINT) VALUE(1) (1)
DCL        VAR(&TO) TYPE(*UINT) VALUE(10)


                                          (2)
DCL        VAR(&DATEN) TYPE(*CHAR) STG(*AUTO) LEN(1000)

DCL        VAR(&POINTER) TYPE(*PTR) (3)

DCL        VAR(&DTAARA) TYPE(*CHAR) STG(*BASED) +
               LEN(100) BASPTR(&POINTER) (4)
                                              (5)
DCL        VAR(&FELD_01) TYPE(*CHAR) STG(*DEFINED) +
               LEN(10) DEFVAR(&DTAARA)

DCL        VAR(&FELD_02) TYPE(*CHAR) STG(*DEFINED) +
               LEN(50) DEFVAR(&DTAARA 11)

DCL        VAR(&FELD_03) TYPE(*CHAR) STG(*DEFINED) +
               LEN(30) DEFVAR(&DTAARA 61)
```

3.3.4
Seite 8

```
DCL        VAR(&FELD_04_C) TYPE(*CHAR) STG(*DEFINED) +
             LEN(5) DEFVAR(&DTAARA 91)
DCL        VAR(&FELD_04_N) TYPE(*DEC) STG(*DEFINED) +
             LEN(5 0) DEFVAR(&DTAARA 91)

DCL        VAR(&FELD_05_1) TYPE(*CHAR) STG(*DEFINED) +
             LEN(5) DEFVAR(&DTAARA 96)
DCL        VAR(&FELD_05_2) TYPE(*CHAR) STG(*DEFINED) +
             LEN(4) DEFVAR(&DTAARA 96)
DCL        VAR(&FELD_05_3) TYPE(*CHAR) STG(*DEFINED) +
             LEN(3) DEFVAR(&DTAARA 98)

RTVDTAARA  DTAARA(DTAARA (1 1000)) RTNVAR(&DATEN) 6

DOFOR      VAR(&VAR) FROM(&FROM) TO(&TO) 7

IF         COND(&VAR = 1) THEN(CHGVAR +
             VAR(&POINTER) VALUE(%ADDR(&DATEN))) 8

ELSE       CMD(CHGVAR VAR(%OFS(&POINTER)) +
             VALUE(%OFS(&POINTER) + 100))

/*   Verarbeitung der jeweils 100 Byte          9 */

            ENDDO
```

Bild 26

1 Definition von Variablen für DOFOR

2 Definition einer Variablen mit 1000 Bytes zum Lesen der Daten aus dem Datenbereich

3 Definition eines Pointer

4 Definition einer Variablen mit 100 Byte. Für diese Variable wird kein Speicherplatz reserviert. Ihre Daten werden jeweils ab der Adresse des basierenden Pointer gefüllt.

5 Definition der Datenstruktur-Unterfelder (Re-Definition der 100 Byte)

6 Einlesen der Daten aus dem Datenbereich

7 Schleife zum Verarbeiten der 10 Segmente

8 Adresse des Pointer bestimmen. Wird pro Schleifendurchlauf – beginnend ab der Adresse der Variablen &DATEN (erster Durchlauf) – jeweils um 100 Byte „verschoben".

   Die Datenstruktur &DTAARA basiert auf dem Pointer. Die Felder/Subfelder werden pro Schleifendurchlauf mit jeweils 100 Byte – beginnend ab der Pointer-Adresse aus dem 1000 Byte langen Feld &DATEN – gefüllt.

9 Hier würde dann die eigentliche Verarbeitung stattfinden.

3.3.4.4 Subroutinen im CL

Ich muss gestehen, mit dieser Neuerung hat mich der Hersteller verblüfft. Subroutine und CL klingt etwas verwegen, ist aber eine geniale Neuerung.

Subroutinen waren für mich schon immer ein probates Mittel, Programme zu strukturieren. EXSR/CAS sind hervorragende „Strukturbefehle".

Die Subroutinen-Befehle im Einzelnen:

- SUBR
- ENDSUBR
- RTNSUBR
- CALLSUBR

Gegenüber den Subroutinen im RPG gibt es einige markante Unterschiede:

1. Subroutinen sind eine Mischung aus klassischer Subroutine und nicht exportierten Subprozeduren.
2. Subprozedur auch deshalb, weil einer Subroutine vom Aufruf zwar keine Eingangsparameter übergeben werden können, aber man kann beim Verlassen einen Rückgabewert definieren und zurückgeben.
3. Es können nur globale Variablen verarbeitet werden. Subprozeduren lassen ja auch lokale Variablen zu.
4. Aus RPG-Subroutinen kann mit GOTO ins Hauptprogramm (Main) verzweigt werden. Ob das ein guter Stil ist, mag dahingestellt sein, aber es ist möglich.

   Ein GOTO in einer Subroutine des CL (sofern überhaupt noch erforderlich) kann nur auf eine Sprungmarke innerhalb der Subroutine verweisen. Verlassen wird die Subroutine durch das Erreichen der ENDSUBR-Anwei-

3.3.4
Seite 10

sung oder die Anweisung RTNSUBR. RTNSUBR ist vergleichbar mit LEAVESR aus RPG oder RETURN aus CL.

5. Subroutinen im CL können rekursiv aufgerufen werden. Jeder Aufruf einer Subroutine erzeugt einen Eintrag im Subroutinen-Stack (vergleichbar mit dem Eintrag von Sub-Prozeduren im Call-Stack). (Siehe dazu auch DCL-PRCOPT).

CL-Subroutinen werden am Ende des Programms, unmittelbar vor der ENDPGM-Anweisung implementiert.

```
                    Subroutine (SUBR)

Auswahl eingeben und Eingabetaste drücken.

Subroutine . . . . . . . . . . . SUBR
```

Name der Subroutine

Bild 27

```
                Subroutine beenden (ENDSUBR)

Auswahl eingeben und Eingabetaste drücken.

Rückkehrwert . . . . . . . . . . RTNVAL        0
```

Rückgabewert (Optional)

Bild 28

```
            Von Subroutine zurückkehren (RTNSUBR)

Auswahl eingeben und Eingabetaste drücken.

Rückkehrwert . . . . . . . . . . RTNVAL        0
```

Verlassen der Subroutine. Der Rückgabewert ist optional.)

Bild 29

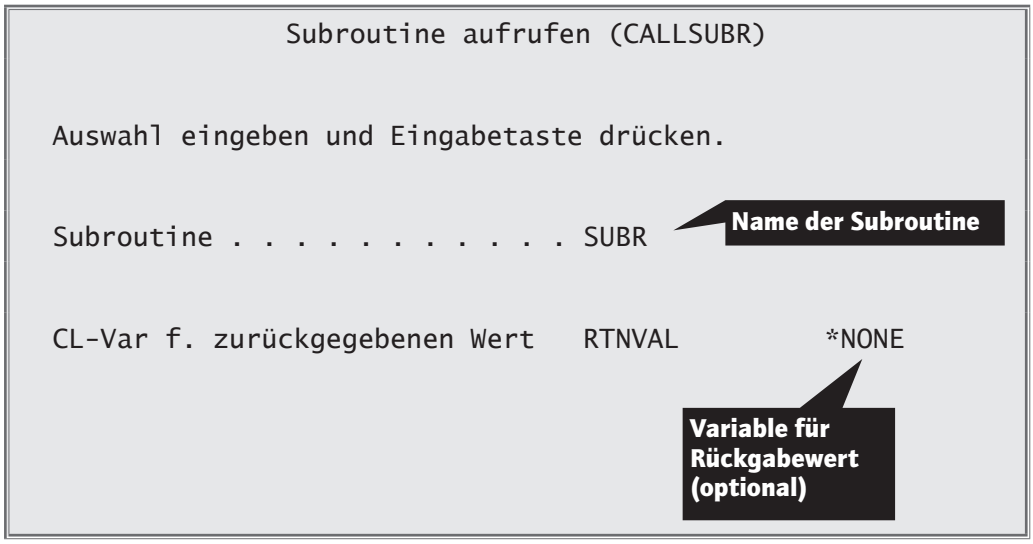


Bild 30

Das Beispielprogramm aus 3.4 wurde so modifiziert, dass einzelne Programmblöcke jetzt in Subroutinen abgehandelt werden. Ich bin der Meinung, dass die Programmstruktur dadurch transparenter geworden ist.

In der eigentlichen Hauptroutine werden nur noch die Subroutinen aufgerufen.

```
   PGM

DCL        VAR(&NBRCURRCD) TYPE(*DEC) LEN(10)
DCL        VAR(&VAR_F01  ) TYPE(*UINT)
DCL        VAR(&TO_F01   ) TYPE(*UINT)
DCL        VAR(&EOF_F02) TYPE(*LGL)
DCL        VAR(&TRUE) TYPE(*LGL) VALUE('1')
DCL        VAR(&FALSE) TYPE(*LGL) VALUE('0')
DCL        VAR(&KZ_MSG) TYPE(*CHAR) LEN(1)

DCLF       FILE(QAFDPHY) OPNID(F01)
DCLF       FILE(QAFDMBR) OPNID(F02)

CALLSUBR   SUBR(FIL_DESC)

CALLSUBR   SUBR(MBR_LIST)
```

Bild 31

3.3.4
Seite 12

Der logische Programmblock zum Verarbeiten der ersten Datei wurde komplett in eine Subroutine übernommen.

```
 /*                                                          */
 /* VERARBEITEN DER DATEIEN IN EINER SUBROUTINE     */
 /*                                                          */
SUBR        SUBR(FIL_DESC)

OVRDBF      FILE(QAFDPHY) TOFILE(##DSPFD)
RTVMBRD     FILE(QTEMP/##DSPFD) NBRCURRCD(&NBRCURRCD)

CHGVAR      VAR(&TO_F01) VALUE(&NBRCURRCD)

OVRDBF      FILE(QAFDPHY) TOFILE(##DSPFD)

DOFOR       VAR(&VAR_F01) FROM(1) TO(&TO_F01) BY(1)
   RCVF        OPNID(F01)

   CHGVAR      VAR(&KZ_MSG) VALUE('1')
   CALLSUBR    SUBR(MSG_OUT)

    /*VERARBEITEN DER DATEN AUS DATEI ##DSPFD         */

            ENDDO

            DLTOVR      FILE(QAFDPHY)
            MONMSG      MSGID(CPF9841)

            ENDSUBR
```

Bild 32

Der logische Programmblock zum Verarbeiten der zweiten Datei wurde komplett in eine Subroutine übernommen.

```
/*                                                                 */
/* VERARBEITEN DER TEILDATEIEN IN EINER SUBROUTINE                 */
/*                                                                 */
            SUBR       SUBR(MBR_LIST)

OVRDBF  FILE(QAFDMBRL) TOFILE(##MEMBR) SHARE(*YES) +
        OPNSCOPE(*ACTGRPDFN)

        CHGVAR     VAR(&EOF_F02) VALUE(&FALSE)

         DOWHILE    COND(&EOF_F02 = &FALSE)       */
           RCVF      OPNID(F02)
           MONMSG    MSGID(CPF0864) EXEC(LEAVE)

           IF        COND(&F02_MLNAME = ' ') THEN(ITERATE)

           CHGVAR     VAR(&KZ_MSG) VALUE('2')
           CALLSUBR   SUBR(MSG_OUT)

           SELECT

            WHEN    COND(&F01_PHACCP = 'K') +
                 THEN(RGZPFM  FILE(&F01_PHLIB/&F01_PHFILE) +
                     MBR(&F02_MLNAME) KEYFILE(*FILE))

         OTHERWISE  CMD(RGZPFM FILE(&F01_PHLIB/&F01_PHFILE) +
                     MBR(&F02_MLNAME))

         ENDSELECT

ENDDO

DLTOVR     FILE(QAFDMBRL)
MONMSG     MSGID(CPF9841)

ENDSUBR
```

Bild 33

3.3.4
Seite 14

Der logische Programmblock zum Ausgeben von Nachrichten wurde komplett in eine Subroutine übernommen.

```
/*                                                                */
/* AUSGABE DER NACHRICHTEN IN SUBROUTINE                          */
/*                                                                */

   SUBR        SUBR(MSG_OUT)

   SELECT

   WHEN     COND(&KZ_MSG = '1') THEN(DO)
      SNDMSG  MSG('Verarbeiten der Datei' *BCAT &F01_PHLIB +
                   *TCAT '/' *TCAT &F01_PHFILE *BCAT 'wird +
                    ausgeführt') TOMSGQ(ITP/REORG_MSG)
   ENDDO

   WHEN        COND(&KZ_MSG = '2') THEN(DO)
       SNDMSG      MSG('Verarbeiten der Teildatei' *BCAT +
                   &F02_MLNAME *BCAT 'wird ausgeführt')  +
                   TOMSGQ(ITP/REORG_MSG)
   ENDDO

   ENDSELECT

   ENDSUBR

   ENDPGM
```

Bild 34

3.3.4.5 Declare Processing Options

Jeder Aufruf einer Subroutine erzeugt einen Eintrag im Subroutine-Stack. Dieser wird entfernt, sobald die Subroutine beendet wird. Wird die Subroutine rekursiv aufgerufen, erzeugt jeder Aufruf einen zusätzlichen Eintrag im Subroutine-Stack.

Die Verschachtelungstiefe für Subroutinen-Aufrufe liegt standardmäßig bei 99. Das sollte in aller Regel ausreichend sein.

Über die CL-Anweisung DCLPRCOPT kann man diesen Wert selbst festlegen.

```
          Verarbeitungsoption deklarieren (DCLPRCOPT)

 Auswahl eingeben und Eingabetaste drücken.

Subroutinenstapeltiefe . . . . . SUBRSTACK      99
```

Bild 35

Der Wertebereich muss zwischen 20 und 9999 liegen. Die Anweisung muss im allgemeinen Deklarationsbereich (am Anfang des Programms) angegeben und darf auch nur einmal definiert werden.

3.3.4.6 Abschluss

Sie sehen, die Control Language hat in der Vergangenheit eine ganze Reihe sinnvoller Erweiterungen erfahren.

Primär denkt man bei Modernisierung an die Vielzahl der RPG/COBOL-Programme und drängt die Steuerprogramme in den Hintergrund.

Dabei steckt auch hier ein ungeheures Potenzial an Möglichkeiten der Modernisierung und damit auch Effizienzsteigerung.

Wenn man bedenkt, wie viele raffinierte Dinge aus CL und Workmanagement nur im Hinterkopf bekannt sind und nicht genutzt werden, kann man manchmal nur erstaunt den Kopf schütteln…

Dazu passend ein chinesisches Sprichwort:

Wenn der Wind der Veränderung weht, bauen die einen Mauern und die anderen Windmühlen!

Nutzen Sie die vielfältigen Möglichkeiten der i5, andere wären froh, wenn Sie nur einen Bruchteil davon hätten.

3.3.4
Seite 16

3.3.5 CL – Neuerungen in V6R1

Der Hersteller hat mit den Neuerungen in der CL-Steuersprache für Release V5R3 und V5R4, was sowohl die Anzahl als auch die Funktionalität betrifft, sehr überrascht. Gemessen an dem Anspruch fallen die Neuerungen für CL in V6R1 doch ziemlich bescheiden aus. Gerade mal drei neue Befehle sind hinzugekommen, funktional aber haben sie einiges zu bieten.

Die Neuerungen im Einzelnen:

3.3.5.1 INCLUDE

In CL-Programmen oder Prozeduren ist es jetzt möglich, mit der INCLUDE-Anweisung zu arbeiten bzw. sie zu verwenden. Was also in anderen Sprachen (RPG usw.) schon lange möglich ist, geht jetzt auch in CL.

INCLUDE ermöglicht es, Source-Code, der in einer anderen Quellen-Teildatei gespeichert ist, während der Kompilierung des eigentlichen CL-Programms/der Prozedur in das Programm/die Prozedur zu kopieren. Damit hat man die Möglichkeit, Code, der in gleicher Form an mehreren Stellen benötigt wird, nur einmal zu codieren. Sofern es sich um eigenständige Routinen handelt, brauchen sie logischerweise auch nur einmal getestet zu werden; bei Änderungen fällt der Aufwand nur einmal an, was enorme Vorteile mit sich bringt. Auch weiterhin bleibt die Möglichkeit bestehen, solche eigenständigen und logisch zusammengehörigen Routinen in ein separates Programm/Prozedur zu „packen“ und dann mit CALL oder CALLPRC aufzurufen. INCLUDE ist eine Erweiterung der Möglichkeiten, die in anderen Sprachen längst gang und gäbe und die nun auch in CL zu verwenden sind.

Beispiel für die Verwendung der INCLUDE-Anweisung:

```
/* einbinden von Code, der in einer anderen Quellenteildatei
*/

/* gespeichert ist, in dieses Programm/Prozedur zur Compile-
zeit */


INCLUDE SRCMBR(V6R1_90) SRCFILE(ITP/QCLSRC)
```

3.3.5
Seite 2

Auszug aus der Umwandlungsliste mit den kopierten Anweisungen:

```
/* START INCLUDE SRCMBR(V6R1_90) SRCFILE(ITP/QCLSRC) */
/* Hier werden nur ein paar Testzeilen definiert, damit man
die */
/* COPY-Anweisung demonstrieren kann. */
/* Text 1 */
/* Text 2 */
/* Text 3 */
/* Text 4 */
/* Text 5 */
/* END INCLUDE SRCMBR(V6R1_90) SRCFILE(ITP/QCLSRC) */
```

In den Quellenanweisungen, die eingebunden werden sollen, darf keine PGM-, keine ENDPGM- und keine weitere INCLUDE-Anweisung enthalten sein.

Sie sollten auch darauf achten, dass, je nachdem wo die INCLUDE Anweisung(en) verwendet wird/werden, alle Deklarationsanweisungen vor den eigentlichen CL-Anweisungen stehen.

3.3.5.2 DCLPRCOPT

Innerhalb der Sprache RPG gibt es die Möglichkeit, in der Steuer- (H-) Anweisung so genannte Umwandlungsanweisungen anzugeben. Das sind Angaben, die eigentlich in der entsprechenden CRT…-Anweisung angegeben werden. Diese Möglichkeit gibt es nun auch in CL. Das hat unbestreitbar Vorteile, denn oft werden solche (notwendigen) Angaben bei der eigentlichen Umwandlung vergessen und führen dann zu fehlerhaften Kompilierungen bzw. zu fehlerhaften Programmausführungen.

Die CL-Anweisung lautet: DCLPRCOPT (Declare Processing Options). Der Befehl selbst darf nur einmal innerhalb einer Quelle angegeben werden (bitte auch beim zuvor vorgestellten INCLUDE beachten) und muss wie alle Deklarationsanweisungen am Anfang des Quellenprogramms stehen. Wenn der Befehl mehrfach verwendet wird, führt das zum Abbruch der Kompilierung (Fehlermeldung CPD0323). Die Anweisungen in der DCLPRCOPT-Anweisung überschreiben (genau wie bei RPG) eventuell anders lautende Angaben beim eigentlichen CRT…-Befehl.

DCLPRCOPT kann sowohl in OPM-Quellen als auch in Prozeduren verwendet werden. Einzelne Werte sind allerdings nur bei OPM erlaubt (z. B. ALWRTVCLSRC), andere nur in ILE-CL-Programmen/Prozeduren (z. B. BNDDIR). Siehe dazu die Spalte „Anmerkungen“ in folgender Tabelle.

Die Reihenfolge der verwendeten Schlüsselwerte innerhalb der DCLPRCOPT-Anweisung ist wahlfrei.

Folgende Angaben/Optionen sind erlaubt:

Keyword	Beschreibung	Mögliche Eingaben/ Unterlassungswerte	Hinweise
SUBRSTACK	Rekursive Aufruftiefe für Subroutinen	20-9999, 99	Optional, Positional 1
LOG	Log Optionen	*JOB, *YES, *NO	Optional
ALWRTVSRC	RTVCLSRC zulassen	*YES, *NO	Nur OPM
TEXT	Text/Beschreibung	*Alphanumerischer Wert,* *SRCMBRTXT, *BLANK	Optional
USRPRF	Benutzer-Profil	*USER, *OWNER	Optional
AUT	Berechtigung	*Name,* *LIBCRTAUT, *CHANGE, *ALL, *USE, *EXCLUDE	Optional
SRTSEQ	Sortierreihenfolge	Einzelne Werte: *HEX, *JOB, *JOBRUN, *LANGIDUNQ, *LANGIDSHR Sonstige Werte: *qualifizierter Objektname*	Optional
	Qualifier 1: Name der Sortiertabelle	Name	
	Qualifier 2: Bibliothek, welche die Sortiertabelle enthält	*Name,* *LIBL, *CURLIB	
LANGID	Sprachen-ID	*Alphanumerischer WeRt,* *JOBRUN, *JOB	Optional
STGMDL	Speichermodell	*SNGLVL, *TERASPACE	Optional
DFTACTGRP	Unterlassungs-Aktivierungsgruppe	*YES, *NO	Optional nur ILE
ACTGRP	Aktivierungsgruppe	*Name,* *STGMDL, *NEW, *CALLER	Optional nur ILE
BNDSRVPGM	Binder-Service-Programm	Einzelne Werte: *NONE Andere Werte (bis zu 300 Serviceprogramme)	Optional nur ILE
	Element 1: Service-Programm	Qualifizierter Objektname	
	Qualifier 1: Service-Programm	*Generischer Name, Name,* *ALL	
	Qualifier 2: Bibliothek	*Name*, *LIBL	
	Element 2: Aktivierung	*IMMED, *DEFER	
BNDDIR	Binderverzeichnis	Einzelne Werte: *NONE andere Werte: (bis zu 300 Binderverzeichnisse)	Optional nur ILE
	Qualifier 1: Binder Verzeichnis Directory	Name	
	Qualifier 2: Bibliothek	*Name,* *LIBL, *CURLIB, *USRLIBL	

Beispiele für die Verwendung einer DCLPRCOPT-Anweisung:

Optionen für OPM-Programm:

```
DCLPRCOPT ALWRTVSRC(*YES) +
USRPRF(*OWNER)
```

Optionen für ILE:

```
DCLPRCOPT BNDDIR(BNDDIR01) +
ACTGRP(ITPXYZ)
```

3.3.5.3 CLOSE

Bereits ab V5R3 war es möglich, mehrere Dateien – maximal fünf – in einem CL-Programm/Prozedur zu deklarieren und zu verarbeiten. Ein entscheidendes Manko blieb bestehen: Jede Datei konnte nur einmal gelesen werden (DB-Dateien sequentiell von BOF bis EOF). Wenn man beispielsweise eine DB-Datei mehrmals im Programm verarbeiten wollte, war das nur durch den Aufruf eines weiteren CL-Programmes möglich. Der OPEN für die Datei erfolgt implizit in CL (RPG lässt grüßen) – und zwar bei der ersten RCVF-Anweisung. SNDRCVF lasse ich an dieser Stelle unberücksichtigt, da damit lediglich Bildschirmdateien (DSPF) verarbeitet werden können. Die mehrmalige Verarbeitung ist hier kein Thema.

An dieser Stelle sei auch ein Hinweis auf den CL-Befehl OPNDBF erlaubt. Dieser öffnet keine Dateien für das CL-Programm, sondern dient lediglich dazu, die Datei für nachgelagerte Anwendungsprogramme (RPG, COBOL) und für „Shared-Open"-Funktionen zu öffnen.

Neu ist ab V6R1 der CL-Befehl CLOSE. CLOSE schließt eine Datei. Ein nachfolgender (erster) RCVF öffnet sie neu und die Verarbeitung beginnt wieder von vorn (BOF).

Beispiele für die Benutzung des neuen CL-Befehls CLOSE:

```
       PGM
/* Deklaration einer DB-Datei nur mit dem Dateinamen */
       DCLF           FILE(SVADP)
/* „Pseudo-Endlosschleife */
       DOWHILEvCOND(1 *EQ 1)
/* lesen der Datei */
       RCVF
/* wenn EOF, Schleife verlassen */
       MONMSG         MSGID(CPF0864) EXEC(LEAVE)
/* Verarbeitung der gelesenen Daten */
       ENDDO
/* Datei schließen für erneutes lesen */
                                  Neu ab V6R1
       CLOSE
/* „Pseudo-Endlosschleife */
       DOWHILE        COND(2 *EQ 2)
/* lesen der Datei */
       RCVF
/* wenn EOF, Schleife verlassen */
       MONMSG         MSGID(CPF0864) EXEC(LEAVE)
/* Verarbeitung der gelesenen Daten */
       ENDDO
       ENDPGM
```

In diesem Beispiel wird nur eine Datei verwendet. Deshalb kann CLOSE ohne weitere Angaben erfolgen. Anschließend wird dann implizit die deklarierte Datei (hier SVADP) angenommen.

Ergänzung 7/2008

3.3.5
Seite 6

```
       PGM
/* Deklaration einer DB-Datei nur mit dem Dateinamen */
       DCLF     FILE(SVADP) OPNID(F01)
/* „Pseudo-Endlosschleife */
       DOWHILE  COND(1 *EQ 1)
/* lesen der Datei */
       RCVF     OPNID(F01)
/* wenn EOF, Schleife verlassen */
       MONMSG   MSGID(CPF0864) EXEC(LEAVE)
/* Verarbeitung der gelesenen Daten */
       ENDDO
/* Datei schließen für erneutes lesen */
       CLOSE    OPNID(F01)
/* „Pseudo-Endlosschleife */
       DOWHILE  COND(2 *EQ 2)
/* lesen der Datei */
       RCVF     OPNID(F01)
/* wenn EOF, Schleife verlassen */
       MONMSG   MSGID(CPF0864) EXEC(LEAVE)
/* Verarbeitung der gelesenen Daten */
       ENDDO
       ENDPGM
```

Arbeiten mit OPNID

Arbeiten mit OPNID

Close mit OPNID

Arbeiten mit OPNID

In diesem Beispiel wird mit OPNID gearbeitet, der dann notwendig ist, wenn mehr als eine Datei im Programm/Prozedur verwendet werden soll (möglich ab V5R3).

3.4 API – Begriffsdefinition

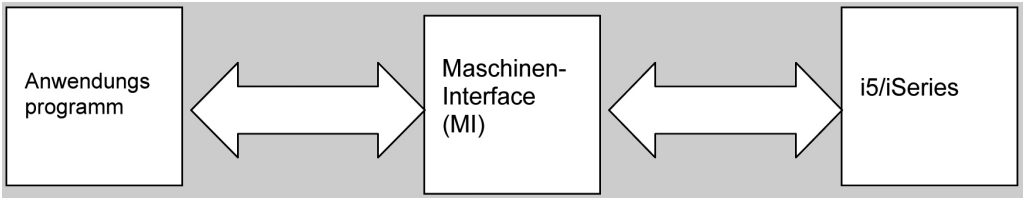
Wikipedia definiert eine „Programmierschnittstelle“ und gibt dazu folgende Erläuterung:

> Eine „Programmierschnittstelle“ ist eine Schnittstelle, die von einem Software-System anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird. Oft wird dafür die Abkürzung API (für Application Programming Interface – deutsch: „Schnittstelle zur Anwendungsprogrammierung“) verwendet. Im Gegensatz zu einer Binärschnittstelle (ABI) definiert API nur die Verwendung der Schnittstellen auf Quelltextebene.

Quelle: Wikipedia

Vereinfacht ausgedrückt heißt das: Hier handelt es sich um ein Systemprogramm, das eine Schnittstelle zwischen Anwendungsprogrammen (ILR RPG, COBOL, CL etc.) und dem System (Hard-/Soft-Ware) herstellt.

Die Kommunikation zwischen den einzelnen Schichten/Komponenten lässt sich wie folgt darstellen:



Anwendungsprogramme sind beispielsweise unsere ILE RPG-Programme und/oder CL-Programme, die ihre Befehle/Bif/SQL-Instruktionen) an das Maschinen-Interface (MI) weiterreichen. Dort werden sie interpretiert, in eine für das System verständliche Weise transformiert und weitergereicht. Auf die gleiche Weise erfolgt die Rückgabe der Informationen an das Programm, wenn sie zum Beispiel einen Kundenstammsatz lesen und die Daten an das Programm übergeben.

Sie schätzen sicher die Vielfalt der Möglichkeiten, die uns die Programmiersprachen – hier speziell die CL-Sprache (über 1.800 Befehle!) – zur Verfügung stellen. Trotzdem gibt es Situationen/Anforderungen, die mit herkömmlichen Mitteln nicht gelöst werden können.

Bestimmt kennen Sie auch die Situation, dass aus einem ILE RPG-Programm heraus ein CL-Befehl ausgeführt werden soll. Die Möglichkeiten dafür sind sicher bekannt: entweder Aufruf eines CL-Programmes das den CL-Befehl ausführt, oder die Ausführung von QCMDEXC, was aber auch ein API ist.

Zusätzlich gibt es eine Reihe von CL-Befehlen, bei denen Sie für die Ausgabe eine sogenannte OUTFILE angeben können, die Sie dann in ihrem Programm

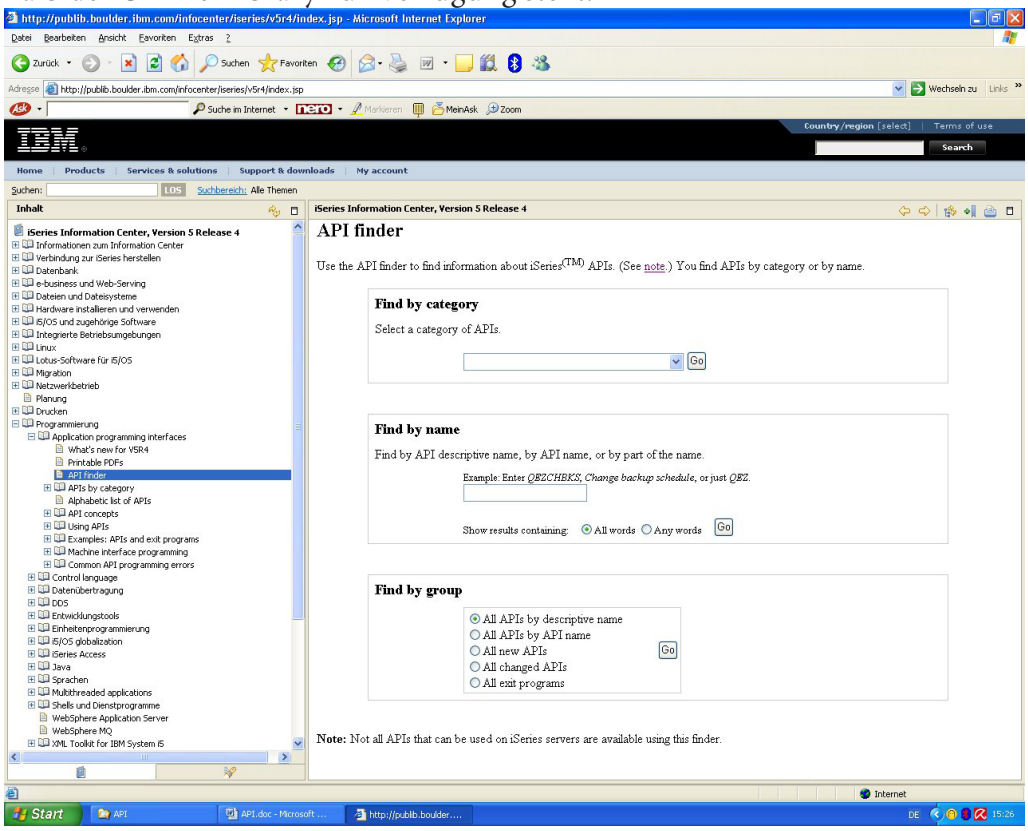
3.4
Seite 2

weiterverarbeiten. Nachteile dieser Variante sind aufwändige Dateioperationen.

Es gibt also eine Reihe von Situationen, bei denen man überlegen muss, ob ein geeignetes API für die Lösung der Aufgabe zur Verfügung steht.

3.4.1 API – Suchen und Finden

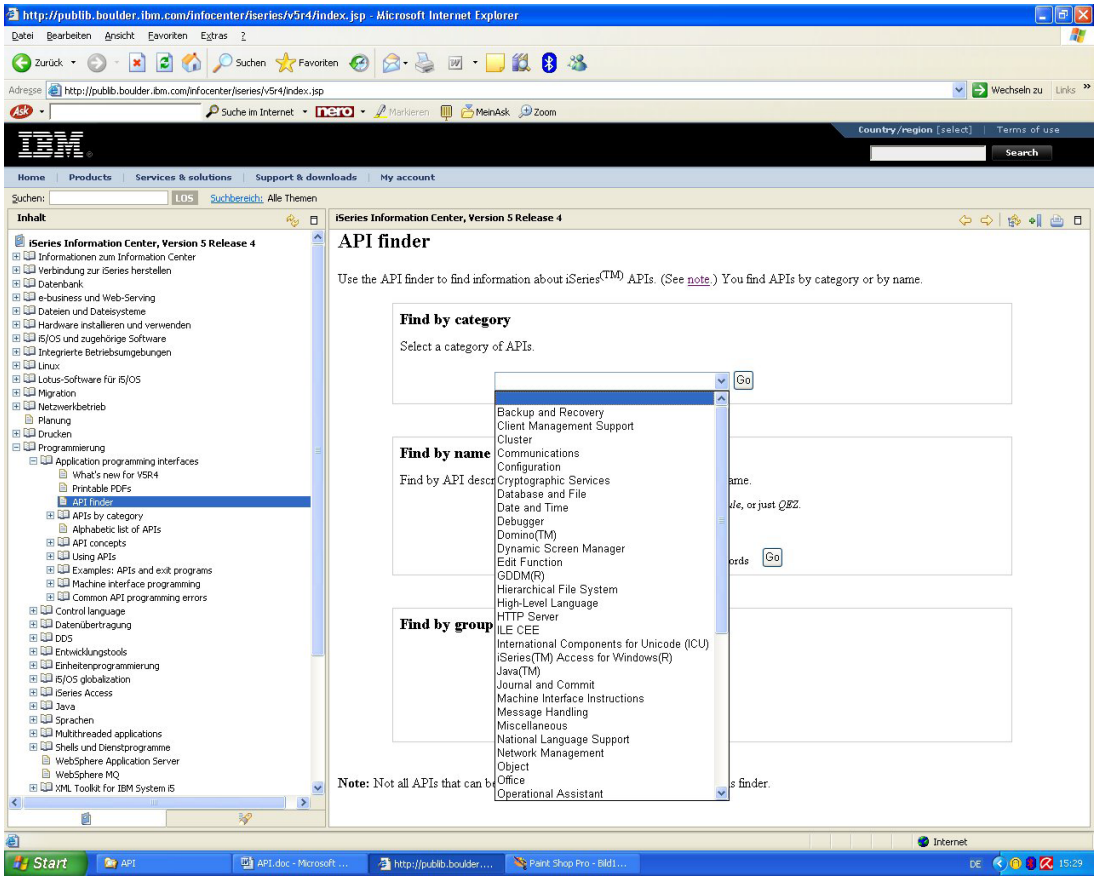
Ein adäquates Bordmittel zum Suchen von APIs ist der API-Finder, der innerhalb der Online-Library zur Verfügung steht.



API-Finder

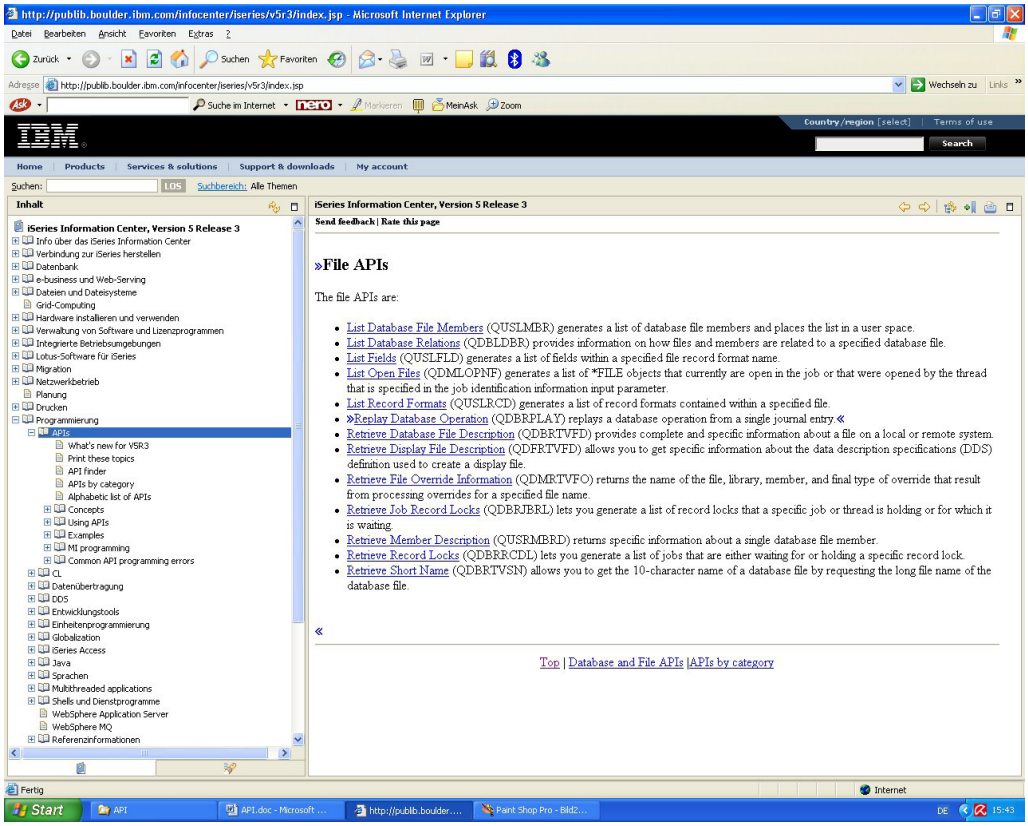
Besonders interessant ist hier die Suchvariante „Select a category of APIs“, wobei natürlich auch die anderen Suchfunktionen nicht abzuwerten sind.

3.4.1
Seite 2



Suchvariante „Select a category of APIs“

Durch die Auswahl (Pfeil nach unten) wird ein Popup-Menü geöffnet. Treffen Sie hier Ihre Auswahl per Mausklick – zum Beispiel: „Database and File“. Daraufhin öffnet sich ein weiteres Fenster. Treffen Sie hier die Auswahl „File APIs“. Nach dieser Auswahl werden Ihnen die APIs zur betroffenen Kategorie angezeigt.



Auswahl „File APIs“

Aus dieser Auswahl werde ich später das API QUSLFLD für ein Beispielprogramm verwenden. Es ist das Äquivalent zum CL-Befehl DSPFFD.

3.4.1
Seite 4

3.4.2 Kommunikation mit APIs

Gemeinsam verwenden alle APIs Standards die definieren, wie man mit diesen Programmen kommuniziert.

Jedes API hat andere Funktionen und stellt damit auch andere Informationen zur Verfügung. Allerdings übertragen alle APIs diese Informationen unter Verwendung identischer Standards.

Die Art, wie diese Informationen zurückgegeben werden unterscheiden sich, aber innerhalb der Art sind sie immer gleich. Manche APIs geben die Informationen in einer einzigen, langen Variablen zurück, andere nutzen dafür einen User Space – zum Beispiel API QUSLFLD, welches in diesem Beitrag vorgestellt wird.

Jedes API hat ein fixes Format, was den Aufruf und die Parameterübergabe betrifft. Diese Parameter sind in der Dokumentation des jeweiligen API beschrieben.

Gemeinsam nutzen viele APIs auch die Art, wie Daten abgerufen werden – nämlich in Form von Formaten. Sie übergeben den Formatnamen an das API und erhalten die Daten in fixen Formaten zurück. Diese Art macht APIs flexibel, was Erweiterungen betrifft. In aller Regel wird dafür ein neues Format definiert. Benötigen Sie also Informationen aus diesen neuen Formaten, müssen Sie logischerweise die Anwendung erweitern. Werden dagegen die neuen Informationen nicht benötigt, so läuft die Anwendung weiter wie gehabt (ohne jegliche Anpassung).

Eine weitere Methode für die Steuerung der Informationsrückgabe ist die Verwendung von Schlüsseln. Auch dazu mehr in den Folgebeiträgen.

3.4.2
Seite 2

3.4.3 Die Bibliothek QSYSINC

Im Normalfall ist auf jeder i5/iSeries eine Bibliothek mit diesem Namen installiert. Falls nicht, kann diese nachträglich installiert werden.

Innerhalb dieser Bibliothek gibt es eine Reihe von Quellendateien – unter anderen auch QRPGLESRC. Die Teildateien enthalten für eine ganze Reihe von APIs Definitionen für die Parameterübergabe beim Aufruf. Außerdem enthalten sie für verschiedene Formate die Beschreibung der Rückgabedaten in Form von Datenstrukturen.

Es lohnt sich, einen Blick in diese Quellendatei zu werfen, obwohl die Feldnamen teilweise recht kryptisch sind und noch aus Zeiten stammen, in denen Feldnamen nur 10 Byte lang sein durften.

Im heutigen Beitrag liegt der Fokus auf List APIs. Sie verwenden zur Ausgabe der Informationen User Spaces. Wie die Verarbeitung von User Spaces erfolgt, erfahren Sie nachfolgend.

Am Beispiel des API QUSLFLD soll dies demonstriert werden. Schauen wir zunächst in die Dokumentation zu diesem API:

List Fields (QUSLFLD) API

```
Required Parameter Group:
1      Qualified user space name  Input       Char(20)
2      Format name                Input       Char(8)
3      Qualified file name        Input       Char(20)
4      Record format name         Input       Char(10)
5      Override processing        Input       Char(1)

Optional Parameter:
6      Error code                 I/O         Char(*)
```

Bei den ersten 5 Parametern handelt es sich um die Übergabeparameter beim Programmaufruf. Auf den optionalen Parameter „Error Code“ werde ich gleich eingehen.

3.4.3
Seite 2

In der Bibliothek QSYSINC – Teildatei QUSLFLD – gibt es folgende Definitionen:

```
D*************************************************************************
D*Type Definition for the Input Parameter Section of the user
D*space in the QUSLFLD API.
D*************************************************************************
DQUSQLI           DS
D*                                              Qdb Lfld Input
D QUSSPACE                1     10
D*                                              Userspace
D QUSULIB                11     20
D*                                              Userspace Library
D QUSFN01                21     28
D*                                              Format Name
D QUSFILN                29     38
D*                                              File Name
D QUSFILLN               39     48
D*                                              File Library Name
D QUSRFN                 49     58
D*                                              Record Format Name
D QUSRRIDE               59     59
D*                                              Override
```

Im dritten Parameter – dem Formatnamen (dieser gibt dem API die Mitteilung, welche Art der Informationen zurückgegeben werden soll) – können folgende Werte angegeben werden:

Format name

INPUT; CHAR(8)

The format of the information returned. You must use the following format name:

FLDL0100	Field information
FLDL0200	Field and default value information
FLDL0300	Field, alternative field name, and default value information

Wenn wir uns für das Format „FLDL0100“ entscheiden, werden folgende Informationen zurück gegeben:

FLDL0100 List Data Section

Offset		Type	Field
Dec	Hex		
0	0	CHAR(10)	Field name
10	A	CHAR(1)	Data type
11	B	CHAR(1)	Use
12	C	BINARY(4)	Output buffer position
16	10	BINARY(4)	Input buffer position
20	14	BINARY(4)	Field length in bytes
24	18	BINARY(4)	Digits
28	1C	BINARY(4)	Decimal position
32	20	CHAR(50)	Field text description
82	52	CHAR(2)	Edit code
84	54	BINARY(4)	Edit word length
88	58	CHAR(64)	Edit word
152	98	CHAR(20)	Column heading 1
172	AC	CHAR(20)	Column heading 2
192	C0	CHAR(20)	Column heading 3
212	D4	CHAR(10)	Internal field name
222	DE	CHAR(30)	Alternative field name
252	FC	BINARY(4)	Length of alternative field name
256	100	BINARY(4)	Number of DBCS characters
260	104	CHAR(1)	Null values allowed
261	105	CHAR(1)	Host variable indicator
262	106	CHAR(4)	Date and time format
266	10A	CHAR(1)	Date and time separator
267	10B	CHAR(1)	Variable length field indicator (overlay for MI mapping)
268	10C	BINARY(4)	Field text description CCSID
272	110	BINARY(4)	Field data CCSID
276	114	BINARY(4)	Field column headings CCSID
280	118	BINARY(4)	Field edit words CCSID
284	11C	BINARY(4)	UCS-2 displayed field length
288	120	BINARY(4)	Field data encoding scheme
292	124	BINARY(4)	Maximum large object field length
296	128	BINARY(4)	Pad length for large object
300	12C	BINARY(4)	Length of user-defined type name
304	130	CHAR(128)	User-defined type name

3.4.3
Seite 4

Offset		Type	Field
Dec	Hex		
432	1B0	CHAR(10)	User-defined type library name
442	1BA	CHAR(1)	Datalink link control
443	1BB	CHAR(1)	Datalink integrity
444	1BC	CHAR(1)	Datalink read permission
445	1BD	CHAR(1)	Datalink write permission
446	1BE	CHAR(1)	Datalink recovery
447	1BF	CHAR(1)	Datalink unlink control
448	1C0	BINARY(4)	Display or print row number
452	1C4	BINARY(4)	Display or print column number
456	1C8	CHAR(1)	ROWID column
457	1C9	CHAR(1)	Identity column
458	1CA	CHAR(1)	GENERATED BY
459	1CB	CHAR(1)	Identity column - CYCLE
460	1CC	DECIMAL(31,0)	Identity column - Original START WITH
476	1DC	DECIMAL(31,0)	Identity column - Current START WITH
492	1EC	BINARY(4)	Identity column - INCREMENT BY
496	1F0	DECIMAL(31,0)	Identity column - MINVALUE
512	200	DECIMAL(31,0)	Identity column - MAXVALUE
528	210	BINARY(4)	Identity column - CACHE
532	214	CHAR(1)	Identity column - ORDER
533	215	CHAR(11)	Reserved

Die Definitionen für dieses Format aus der Bibliothek QSYSINC, Teildatei QUSLFLD:

```
D*****************************************************************
 D*Type Definition for the FLDL0100 format of the user space in the
 D*QUSLFLD API.                                                §A2C
 D*****************************************************************
 DQUSL0100         DS
 D*                                          Qdb Lfld FLDL0100
 D QUSFN02                 1     10
 D*                                          Field Name
 D QUSDT                  11     11
 D*                                          Data Type

 D QUSU                   12     12
```

```
D*                                        Use
D QUSOBP                 13     16B 0
D*                                        Output Buffer Position
D QUSIBP                 17     20B 0
D*                                        Input Buffer Position
D QUSFLB                 21     24B 0
D*                                        Field Length Bytes
D QUSIGITS               25     28B 0
D*                                        Digits
D QUSDP                  29     32B 0
D*                                        Decimal Positions
D QUSFTD                 33     82
D*                                        Field Text Description
D QUSEC00                83     84
D*                                        Edit Code
D QUSEWL                 85     88B 0
D*                                        Edit Word Length
D QUSEW                  89    152
D*                                        Edit Word
D QUSCH1                153    172
D*                                        Column Heading1
D QUSCH2                173    192
D*                                        Column Heading2
D QUSCH3                193    212
D*                                        Column Heading3
D QUSIFN                213    222
D*                                        Internal Field Name
D QUSAFN                223    252
D*                                        Alternate Field Name
D QUSLAF                253    256B 0
D*                                        Length Alternate Field
D QUSDBCSC              257    260B 0
D*                                        Number DBCS Characters
D QUSNVA                261    261
D*                                        Null Values Allowed
D QUSVFI                262    262
D*                                        Variable Field Indicator
D QUSDTF                263    266
```

3.4.3
Seite 6

```
D*                                        Date Time Format
D QUSDTS                 267    267
D*                                        Date Time Separator
D QUSVLFI                268    268
D*                                        Variable Length Field Ind
D QUSCCSID00             269    272B 0
D*                                        Field Description CCSID
D QUSCCSID01             273    276B 0
D*                                        Field Data CCSID
D QUSCCSID02             277    280B 0
D*                                        Field Column Heading CCSID
D QUSCCSID03             281    284B 0
D*                                        Field Edit Words CCSID
D QUSCS2DL               285    288B 0
D*                                        UCS2 Displayed Length
D QUSFDES                289    292B 0
D*                                        Field Data Encoding Scheme
D QUSMLOL                293    296B 0
D*                                        Maximum Large Object Length
D QUSLOPL                297    300B 0
D*                                        Large Object Pad Length
D QUSUDTNL               301    304B 0
D*                                        UDT Name Length
D QUSUDTN                305    432
D*                                        UDT Name
D QUSUDTLN               433    442
D*                                        UDT Library Name
D QUSDC03                443    443
D*                                        Datalink Control
D QUSDI03                444    444
D*                                        Datalink Integrity
D QUSDRP                 445    445
D*                                        Datalink Read Permission
D QUSDWP                 446    446

D*                                        Datalink Write Permission
D QUSDR                  447    447
D*                                        Datalink Recovery
D QUSDUC02               448    448
```

```
D*                                        Datalink Unlink Control
D QUSRNBR               449    452B 0
D*                                        Row Number
D QUSCNBR               453    456B 0
D*                                        Column Number
D QUSOWIDC              457    457
D*                                        ROWID Column
D QUSIC                 458    458
D*                                        Identity Column
D QUSTEDBY              459    459
D*                                        GENERATED BY
D QUSICC                460    460
D*                                        IC Cycle
D QUSICOSW              461    476P 0
D*                                        IC Orig Start With
D QUSICCSW              477    492P 0
D*                                        IC Curr Start With
D QUSICIB               493    496B 0
D*                                        IC Increment By
D QUSVALUE              497    512P 0
D*                                        IC MINVALUE
D QUSVALUE00            513    528P 0
D*                                        IC MAXVALUE
D QUSCACHE              529    532U 0
D*                                        IC CACHE
D QUSORDER              533    533
D*                                        IC ORDER
D QUSERVED47            534    544
D*                                        Reserved
```

Die Ausgabe der Daten erfolgt in einen User Space.

3.4.3
Seite 8

3.4.4 Der Aufruf von APIs

Um ein API zu verwenden (aufzurufen), reicht in der Regel ein CALL. Die Parameterübergabe erfolgt auf die gleiche Weise wie beim Aufruf eines anderen Anwendungsprogramms. Das bedeutet: Bei der Parameterübergabe erfolgt kein regelrechter Datenaustausch, sondern es werden nur Pointer (Adressen) zwischen den Programmen ausgetauscht, die auf die Daten verweisen. Man spricht hierbei von „Übergabe von Parametern nach Verweis“ („Passing Parameters by Reference“).

Manche APIs verlangen explizit (Dokumentation!) die Übergabe von echten Daten. Diese Methode wird als „Übergabe von Parametern nach Wert“ („Passing Parameters by Value“) bezeichnet. Wenn das erforderlich ist, müssen Sie beim jeweiligen Parameter das Schlüsselwort *VALUE angeben.

Bei vielen APIs fällt auf, dass Parameter „optional“ sind (siehe oben: „Error Code“).

Wenn ein Parameter optional ist, sind alle nachfolgenden Parameter ebenfalls optional. Das bedeutet, dass Sie, wenn Sie einen optionalen Parameter weglassen, alle nachfolgenden, ebenfalls optionalen Parameter auch weglassen müssen.

Dies ist nicht zu verwechseln mit Parametern die nicht übergeben werden müssen und somit bei der Übergabe „weggelassen" werden können. Parameter die weggelassen werden, müssen trotzdem mit einem Null-Pointer übergeben werden. Man erreicht dies durch die Angabe des Schlüsselworts *OMIT beim jeweiligen Parameter. Das bedeutet allerdings wiederum, dass Parameter, die auf „weggelassene“ Parameter folgen, trotzdem verwendet werden können.

3.4.4 Seite 2

3.4.5 Datentypen bei der Verwendung von APIs

Wenn APIs numerische Daten verwenden, handelt es sich hierbei fast immer um binäre Zahlen. In diesem Zusammenhang möchte ich auf das Kapitel 3.1.10.4.1 verweisen, in dem auf diese Datenart hingewiesen wird.

Ein weiterer Datentyp sind Pointer. Alle Pointer sind laut Definitionen 16 Byte lang. Anders als bei üblichen Datentypen beschreibt ein Pointer nicht wirklich ein Datenelement, vielmehr zeigt er auf eine Adresse, die die eigentlichen Daten enthält. Gerade bei der Verarbeitung von User Spaces wird sehr oft mit Pointern gearbeitet.

Pointer können, müssen aber nicht definiert werden. Dazu folgendes Beispiel:

```
D Pointer_1       s                 *    1

D DatenDS         ds                     based(Pointer_2)   2
D  DatenDS_01                    10
D  DatenDS_02                     5s 0
```

1 Pointer_1 ist über die Datenart „*“ explizit als Pointer definiert.

2 Pointer_2 ist nirgendwo (explizit) im Programm definiert, wird aber automatisch (implizit) durch die Angabe des Schlüsselwortes „based“ definiert.

Pointer haben keine feste Speicheradresse. Erst durch das Setzen einer Pointer-Adresse wird die Anzahl der definierten Bytes ab der angegebenen Adresse verarbeitet.

3.4.5
Seite 2

3.4.6 Der optionale Parameter „Error-Code“

In fast allen APIs kann dieser Parameter angegeben werden. Er informiert über eventuell auftretende Fehler bei der API-Ausführung und sollte deshalb grundsätzlich verwendet werden. Bei einigen APIs gibt es ein zweites Error-Format, auf dessen Beschreibung ich zunächst verzichte.

Der Parameter selbst ist eine Datenstruktur mit folgendem Aufbau:

Format ERRC0100

Offset		Use	Type	Field
Dec	Hex			
0	0	INPUT	BINARY(4)	Bytes provided
4	4	OUTPUT	BINARY(4)	Bytes available
8	8	OUTPUT	CHAR(7)	Exception ID
15	F	OUTPUT	CHAR(1)	Reserved
16	10	OUTPUT	CHAR(*)	Exception data

Auch dafür gibt es Definitionen in der Bibliothek QSYSINC in der Teildatei QUSEC in Form einer Datenstruktur:

```
D*Record structure for Error Code Parameter
D*NOTE: The following type definition only defines the fixed
D*   portion of the format.  Verying length field Exception
D*   Data will not be defined here.
D*****************************************************************
DQUSEC            DS
D*                                          Qus EC
D QUSBPRV                 1      4B 0
D*                                          Bytes Provided
D QUSBAVL                 5      8B 0
D*                                          Bytes Available
D QUSEI                   9     15
D*                                          Exception Id
D QUSERVED               16     16
D*                                          Reserved
D*QUSED01                17     17
D*
D*                                    Varying length
```

3.4.6
Seite 2

Der letzte Parameter enthält im Fehlerfall den Text der Fehlernachricht. Obwohl dieses Feld mit einer variablen Länge angegeben ist, genügt es fast immer, hier mit einer Feldlänge von 132 Byte zu arbeiten (maximale Länge von Nachrichtentexten der ersten Ebene). Aus diesem Grund und dem Wunsch von eigenen Feldnamen habe ich selbst eine Definition erstellt.

```
D  ErrorDS        ds
d   BytesProv                   10i 0 inz(%size(ErrorDs))
d   BytesAvail                  10i 0
d   MsgId                        7a
d   Reserviert                   1a
d   ErrorMsg                   132a
```

3.4.7 API QUSLFLD

Wie bereits erwähnt, ist dieses API ein Äquivalent zum CL-Befehl DSPFFD. Es ist damit unter anderem geeignet, direkt in einem Programm Satzaufbauten von Dateien abzubilden (zum Beispiel für eigene Dokumentationsprogramme).

Die Ausgabe erfolgt in einen User Space.

3.4.7.1 APIs für User Spaces

Ein User Space ist eine eigene Objektart von i5/OS. Für die Verarbeitung (Erstellen, Lesen, Löschen usw.) stehen wiederum APIs zur Verfügung.

Nun wird QUSLFLD sicher nicht die einzige API sein, die Sie verwenden wollen und die Daten in einen User Space ausgibt. Deshalb sollten Sie sich Gedanken darüber machen, ob für Standardaufgaben in Verbindung mit dem User Space Standardroutinen beziehungsweise Services erstellt werden, die die ‚lästige' Arbeit erledigen und die diese als Ergebnis in einem Serviceprogramm zur Verfügung stellen. Damit wird repetitiver Code vermieden.

Für die Erstellung des User Space benötigen wir das API QUSCRTUS. Da der User Space über Pointer verarbeitet werden soll, muss auf den erstellten User Space ein Pointer „gesetzt" werden. Dies geschieht mit dem API QUSPTRUS.

Wenn ein User Space erstellt wird, kann als maximale Länge 16 MB angegeben werden Es ist sicher nicht sinnvoll, den User Space grundsätzlich mit dieser maximalen Länge zu erstellen. Besser ist es, eine sinnvolle Anfangsgröße zu wählen und anschließend den User Space bei Bedarf zu erweitern. Dazu muss nach dem Erstellen des User Space ein entsprechendes Attribut gesetzt werden (die eigentliche Erweiterung erfolgt dann bei Bedarf automatisch). Diese Attributänderung erfolgt mit dem API QUSCUSAT.

Ergänzung 2/2007

3.4.7
Seite 2

Create User Space (QUSCRTUS) API

1	Qualified user space name	Input	Char(20)
2	Extended attribute	Input	Char(10)
3	Initial size	Input	Binary(4)
4	Initial value	Input	Char(1)
5	Public authority	Input	Char(10)
6	Text description	Input	Char(50)
Optional Parameter Group 1:			
7	Replace	Input	Char(10)
8	Error code	I/O	Char

Retrieve Pointer to User Space (QUSPTRUS) API

Required Parameter Group:			
1	Qualified user space name	Input	Char(20)
2	Return pointer	Output	PTR(SPP)
Optional Parameter:			
3	Error code	I/O	Char(*)

Change User Space Attributes (QUSCUSAT) API

Required Parameter Group:			
1	Returned library name	Output	Char(10)
2	Qualified user space name	Input	Char(20)
3	Attributes to change	Input	Char(*)
4	Error code	I/O	Char(*)

Der Code des Serviceprogramms (bitte Kommentare beachten):

```
  *
*** Das Modul enthält keine Hauptprozedur
*
 nomain
*
*** Definition Prototyp für das Serviceprogramm
*
 CrtUsrSpc       pr              *

                                20    const
*
*** Beginn der Prozedur. Die Prozedur kann extern aufgerufen werden
*
 CrtUsrSpc       b                   export
*
*** Definition Prozedurenschnittstelle (Parameter)
*** Der Rückgabeparameter ist ein Pointer auf den User Space

*** Als Eingangsparameter werden der Name des User Space und die Bibliothek
*** übergeben (in einem Feld mit jeweils 10 Stellen Länge). Dieses Feld
*** ist schreibgeschützt (CONST).
 *
D CrtUsrSpc       PI              *
D  UsrSpcName                   20    const
 *
 *** Re-Definition des Eingangsparameters
 *
D  SpnamBib       ds
d   SpaceName                   10
d   SpaceBibl                   10
 *
 *** Definition Error-Datenstruktur
 *
D  ErrorDS        ds
d   BytesProv                   10i 0 inz(%size(ErrorDs))
d   BytesAvail                  10i 0
d   MsgId                        7a
```

3.4.7
Seite 4

```
d   Reserviert                  1a
d   ErrorMsg                  132a
 *
 *** Angaben für zum API QUSCUSAT
 *
D  ChgUsrSpc      ds

 *
 *** Anzahl der Keys

 *
d   NbrAttr                    9b 0  inz(1)
 *
 *** Dieses Attribut wird über den Schlüssel geändert
 *
d   Key                        9b 0  inz(3)
 *
 *** Länge der Daten (nächster Wert)
 *
d   Length                     9b 0  inz(1)
 *
 *** Durch die Angabe von 1 wird der User Space auf automatisch änderbar
gesetzt
 *
d   Value                      1     inz('1')
 *
 *** Pointer-Definition für Rückgabe
 *
d SpcPtr          s               *
 *
 *** Parameter für QUSRCRTUS
 *
d ErrortAttr      s             10
d SpcExtAttr      s             10
d SpcLength       s              9b 0 inz(2048)
d SpcValue        s              1    inz(X'00')
d SpcAuth         s             10    inz(,*CHANGE')
d SpcText         s             50    inz('allgemeiner User Space')
d SpcReplace      s             10    INZ(,*YES')
```

```
c*
c*** Übertragene Eingangsparameter
c*
c                   eval      SpnamBib = UsrSpcName
c*
c*** Erstellen User Space
c*
c                   call      ,QUSCRTUS‘
c                   parm                    SpnamBib
c                   parm                    SpcExtAttr
c                   parm                    SpcLength
c                   parm                    SpcValue
c                   parm                    SpcAuth
c                   parm                    SpcText

c                   parm                    SpcReplace
c                   parm                    ErrorDs
c*

c*** Abruf Pointer für User Space
c*
c                   call      ,QUSPTRUS‘
c                   parm                    SpnamBib
c                   parm                    SpcPtr
c*
c*** Ändern Attribute des User Space
c*
c                   call      ‘QUSCUSAT’
c                   parm                    SpaceBibl
c                   parm                    SpnamBib
c                   parm                    ChgUsrSpc
c                   parm                    ErrorDs
c*
c*** Programm Ende
c*
c                   return     Spcptr
c*
p CrtUsrSpc       e
```

3.4.7
Seite 6

3.4.7.2 Gemeinsame Datenstrukturen von List APIs im User Space

Alle List APIs, die einen User Space verwenden, haben am Anfang eine allgemeine Datenstruktur, die auch als „General Data Structure" bezeichnet wird. Dieser Bereich hat einen fest definierten Aufbau und ist für alle APIs gleich.

Allgemeine Datenstruktur (General Data Structure)
64 Byte Benutzerbereich
Größe der Header-Datenstruktur
Die eigentliche Header-Datenstruktur

Für diese Datenstruktur gibt es in der Bibliothek QSYSINC in der Datei QRPGLESRC ebenfalls eine vordefinierte Datenstruktur in der Teiledatei QUSGEN:

```
D*****************************************************************
D*Type Definition for the User Space Generic Header, 300 format.
D*****************************************************************
DQUSH0300         DS
D*                                          Qus Generic Header 0300
D QUSUA00                 1     64
D*                                          User Area
D QUSSGH00               65     68B 0
D*                                          Size Generic Header
D QUSSRL00               69     72
D*                                          Structure Release Level
D QUSFN00                73     80
D*                                          Format Name
D QUSAU00                81     90
D*                                          Api Used
D QUSDTC00               91    103
D*                                          Date Time Created
D QUSIS00               104    104
D*                                          Information Status
D QUSSUS00              105    108B 0
D*                                          Size User Space
D QUSOIP00              109    112B 0
D*                                          Offset Input Parameter
```

```
D QUSSIP00              113    116B 0
D*                                          Size Input Parameter
D QUSOHS00              117    120B 0
D*                                          Offset Header Section
D QUSSHS00              121    124B 0
D*                                          Size Header Section
D QUSOLD00              125    128B 0
D*                                          Offset List Data
D QUSSLD00              129    132B 0
D*                                          Size List Data
D QUSNBRLE00            133    136B 0
D*                                          Number List Entries
D QUSSEE00              137    140B 0
D*                                          Size Each Entry
D QUSSIDLE00            141    144B 0
D*                                          CCSID List Ent
D QUSCID00              145    146
D*                                          Country ID
D QUSLID00              147    149
D*                                          Language ID
D QUSSLI00              150    150
D*                                          Subset List Indicator
D QUSRSV1               151    192
D*                                          Reserved 1
D QUSEPN                193    448
D*                                          Entry Point Name
D QUSRSV2               449    576
D*                                          Reserved 2
```

Verwenden Sie für ILE RPG das Format 0300; das Format 0100 gilt ausschließlich für OPM-Programme.

Die hauptsächliche Funktion der Generic Header-Datenstruktur ist das Bereitstellen von Informationen über den Inhalt (der Daten) im restlichen Benutzerbereich.

Primär benötigt man die Informationen, ab welcher Stelle im User Space die Sektion der List-Daten (List Data Section) beginnt, wie viele List-Einträge der User Space enthält und wie lang (Anzahl Byte) jeder List-Eintrag ist.

3.4.7
Seite 8

Wir werden im ILE RPG-Programm diese Daten über Pointer in einer Schleife verarbeiten. Dazu benötigen wir aus obiger Datenstruktur die Subfelder:

- QUSOLD00 Beginn der List-Daten
- QUSNBRLE00 Anzahl der List-Einträge
- QUSSEE00 Länge jedes List-Eintrags

In der Beschreibung der Datenstruktur wird in diesem Zusammenhang von Offset gesprochen, was für uns „Zeiger = Pointer“ bedeutet. Wichtig ist es, zu wissen, dass ein Offset immer bei einem Wert 0 beginnt. Wir müssen daher den zur Verfügung gestellten Wert in Wirklichkeit um ein 1 erhöhen.

Ergänzt sei noch, dass die einzelnen APIs spezifische Informationen in einer Input-Parameter-Sektion und in einer Header-Sektion zur Verfügung stellen. Falls diese Daten verarbeitet werden, stehen dafür in der obigen Datenstruktur (Header-Datenstruktur) ebenfalls Sub-Felder zur Verfügung.

Wichtig ist auch die Information, dass einige aufwändigere List APIs diese Felder nicht verwenden. Diese Tatsache soll uns aber an dieser Stelle nicht interessieren.

Ich habe das Programm so gestaltet, dass es eine bestimmte Datei fix verarbeitet. In der Praxis würde man das sicher variabel gestalten (Parameter) und es so ergänzen, dass mit den Daten aus der API-Verwendung auch konkret eine Verarbeitung stattfindet. Darauf habe ich hier verzichtet, da es nur um die Funktion „API“ geht.

Im Folgenden sehen Sie das Beispiel für API QUSLFLD. Bitte beachten Sie auch hier wieder die Kommentare. Die Parameter aus der Dokumentation sind bereits oben beschrieben.

```
 *
  *** Definition Prototyp für Aufruf der Prozedur zum Erstellen des User Space
  *
D CrtUsrSpc       pr              *
d                                 20    const
  *
  *** Name und Bibliothek für den User Space
  *
D UserSpace       s               20    inz('USRSPACE  QTEMP     ')
  *
  *** Definition Parameter-Felder für die API QUSLFLD
  *
D #FmtName        s                8    inz('FLDL0100')
D #FilBibl        s               20    inz('SVADP     ITP       ')
```

```
D #RecFmtN        s             10    inz(,SVADF     ,)
D #Overrid        s              1    inz(,0‘)
 *
 *** Arbeitsfelder
 *
D #ix             s              5s 0
 *
 *** Pointer Definition für Aufruf   von CrtUsrSpc
 *

d SpcPtr          s               *
 *
 *** Definition Error-Datenstruktur für API-Aufrufe
 *
D  ErrorDS        ds
d   BytesProv                   10i 0 inz(%size(ErrorDs))
d   BytesAvail                  10i 0
d   MsgId                        7a
d   Reserviert                   1a
d   ErrorMsg                   132a
 *
 *
 *** Definition der Datenstruktur für die List-Daten - übernommen von QSYSINC
 *** und definieren BASED Pointer
 *
D*****************************************************************
D*Type Definition for the FLDL0100 format of the user space in the
D*QUSLFLD API.                                                §A2C
D*****************************************************************
DQUSL0100         DS                          Based(LstPtr)
D*                                            Qdb Lfld FLDL0100
D QUSFN02                1     10
D*                                            Field Name
D QUSDT                 11     11
D*                                            Data Type
D QUSU                  12     12
D*                                            Use
D QUSOBP                13     16B 0
D*                                            Output Buffer Position
```

3.4.7
Seite 10

```
D QUSIBP                 17     20B 0
D*                                     Input Buffer Position
D QUSFLB                 21     24B 0
D*                                     Field Length Bytes
D QUSIGITS               25     28B 0
D*                                     Digits
D QUSDP                  29     32B 0
D*                                     Decimal Positions
D QUSFTD                 33     82
D*                                     Field Text Description
D QUSECOO                83     84
D*                                     Edit Code
D QUSEWL                 85     88B 0
D*                                     Edit Word Length
D QUSEW                  89    152
D*                                     Edit Word
D QUSCH1                153    172
D*                                     Column Heading1
D QUSCH2                173    192
D*                                     Column Heading2
D QUSCH3                193    212
D*                                     Column Heading3
D QUSIFN                213    222
D*                                     Internal Field Name
D QUSAFN                223    252
D*                                     Alternate Field Name
D QUSLAF                253    256B 0
D*                                     Length Alternate Field
D QUSDBCSC              257    260B 0
D*                                     Number DBCS Characters
D QUSNVA                261    261
D*                                     Null Values Allowed
D QUSVFI                262    262
D*                                     Variable Field Indicator
D QUSDTF                263    266
D*                                     Date Time Format
D QUSDTS                267    267
D*                                     Date Time Separator
```

```
D QUSVLFI               268    268
D*                                          Variable Length Field Ind
D QUSCCSID00            269    272B 0
D*                                          Field Description CCSID
D QUSCCSID01            273    276B 0
D*                                          Field Data CCSID
D QUSCCSID02            277    280B 0
D*                                          Field Column Heading CCSID
D QUSCCSID03            281    284B 0
D*                                          Field Edit Words CCSID
D QUSCS2DL              285    288B 0
D*                                          UCS2 Displayed Length
D QUSFDES               289    292B 0
D*                                          Field Data Encoding Scheme
D QUSMLOL               293    296B 0
D*                                          Maximum Large Object Length
D QUSLOPL               297    300B 0
D*                                          Large Object Pad Length
D QUSUDTNL              301    304B 0
D*                                          UDT Name Length
D QUSUDTN               305    432
D*                                          UDT Name
D QUSUDTLN              433    442
D*                                          UDT Library Name
D QUSDC03               443    443
D*                                          Datalink Control
D QUSDI03               444    444
D*                                          Datalink Integrity
D QUSDRP                445    445
D*                                          Datalink Read Permission
D QUSDWP                446    446
D*                                          Datalink Write Permission
D QUSDR                 447    447
D*                                          Datalink Recovery
D QUSDUC02              448    448
D*                                          Datalink Unlink Control
D QUSRNBR               449    452B 0
D*                                          Row Number
```

3.4.7
Seite 12

```
D QUSCNBR               453    456B 0
D*                                          Column Number
D QUSOWIDC              457    457
D*                                          ROWID Column
D QUSIC                 458    458
D*                                          Identity Column
D QUSTEDBY              459    459
D*                                          GENERATED BY
D QUSICC                460    460
D*                                          IC Cycle
D QUSICOSW              461    476P 0
D*                                          IC Orig Start With
D QUSICCSW              477    492P 0
D*                                          IC Curr Start With
D QUSICIB               493    496B 0
D*                                          IC Increment By
D QUSVALUE              497    512P 0
D*                                          IC MINVALUE
D QUSVALUE00            513    528P 0
D*                                          IC MAXVALUE
D QUSCACHE              529    532U 0
D*                                          IC CACHE
D QUSORDER              533    533
D*                                          IC ORDER
D QUSERVED47            534    544
D*                                          Reserved
 *
 *** Definition der General Header Datenstruktur (Format 0300)

 *** - übernommen von QSYSINC  und definieren BASED Pointer
 *
D*****************************************************************

D*Type Definition for the User Space Generic Header, 300 format.
D*****************************************************************
DQUSH0300         DS                        based(HdrPtr)
D*                                          Qus Generic Header 0300
D QUSUA00                1     64
D*                                          User Area
```

```
D QUSSGH00                65     68B 0
D*                                      Size Generic Header
D QUSSRL00                69     72
D*                                      Structure Release Level
D QUSFN00                 73     80
D*                                      Format Name
D QUSAU00                 81     90
D*                                      Api Used
D QUSDTC00                91    103
D*                                      Date Time Created
D QUSIS00                104    104
D*                                      Information Status
D QUSSUS00               105    108B 0
D*                                      Size User Space
D QUSOIP00               109    112B 0
D*                                      Offset Input Parameter
D QUSSIP00               113    116B 0
D*                                      Size Input Parameter
D QUSOHS00               117    120B 0
D*                                      Offset Header Section
D QUSSHS00               121    124B 0
D*                                      Size Header Section
D QUSOLD00               125    128B 0
D*                                      Offset List Data
D QUSSLD00               129    132B 0
D*                                      Size List Data
D QUSNBRLE00             133    136B 0
D*                                      Number List Entries
D QUSSEE00               137    140B 0
D*                                      Size Each Entry
D QUSSIDLE00             141    144B 0
D*                                      CCSID List Ent
D QUSCID00               145    146
D*                                      Country ID
D QUSLID00               147    149
D*                                      Language ID
D QUSSLI00               150    150
D*                                      Subset List Indicator
```

3.4.7
Seite 14

```
D QUSRSV1               151    192
D*                                          Reserved 1
D QUSEPN                193    448
D*                                          Entry Point Name
D QUSRSV2               449    576
D*                                          Reserved 2
c*
c*** Aufruf Prozedur zum Erstellen des User Space
c*
c                   eval      Spcptr = CrtUsrSpc(UserSpace)
c*
c*** Aufruf API QUSLFLD
c*
C                   call      ,QUSLFLD‘
C                   parm                    UserSpace
C                   parm                    #FmtName
C                   parm                    #FilBibl
C                   parm                    #RecFmtN
C                   parm                    #Overrid
C                   parm                    ErrorDs
c*
c*** Bereitstellen Daten der allgemeinen Datenstruktur
c*
c                   eval       HdrPtr = SpcPtr
c*
c*** Bereitstellen Listendaten
c*
c                   eval       LstPtr = HdrPtr + QusOLDOO
c*
c*** Schleife zum Verarbeiten der Listeinträge
c*
c                   for        #ix = 1 to QusNbrLEOO
c*
c*** an dieser Stelle würde die Verarbeitung der Daten erfolgen
c*
c*
c*** Pointer auf nächstes Listenelement verschieben
c*
```

```
c                   eval       LstPtr = LstPtr  + QusSEE00
c                   endfor
c*
c*** Programmende
c*
c                   eval       *inlr = *on
```

3.4.7
Seite 16