software AG

Natural für Windows

Leitfaden zur Programmierung

Version 6.3.8 für Windows

Februar 2010

Natural

Dieses Dokument gilt für Natural ab Version 6.3.8 für Windows.

Hierin enthaltene Beschreibungen unterliegen Änderungen und Ergänzungen, die in nachfolgenden Release Notes oder Neuausgaben bekanntgegeben werden.

Copyright © 1992-2010 Software AG, Darmstadt, Deutschland und/oder Software AG USA, Inc., Reston, VA, Vereinigte Staaten von Amerika, und/oder ihre Lizenzgeber..

Der Name Software AG, webMethods und alle Software AG Produktnamen sind entweder Warenzeichen oder eingetragene Warenzeichen der Software AG und/oder der Software AG USA, Inc und/oder ihrer Lizenzgeber. Andere hier erwähnte Unternehmens- und Produktnamen können Warenzeichen ihrer jeweiligen Eigentümer sein.

Die Nutzung dieser Software unterliegt den Lizenzbedingungen der Software AG. Diese Bedingungen sind Bestandteil der Produktdokumentation und befinden sich unter http://documentation.softwareag.com/legal/ und/oder im Wurzelverzeichnis des lizensierten Produkts.

Diese Software kann Teile von Drittanbieterprodukten enthalten. Die Hinweise zu den Urheberrechten und Lizenzbedingungen der Drittanbieter entnehmen Sie bitte den "License Texts, Copyright Notices and Disclaimers of Third Party Products". Dieses Dokument

ist Bestandteil der Produktdokumentation und befindet sich unter http://documentation.softwareag.com/legal/ und/oder im Wurzelverzeichnis des lizensierten Produkts.

Inhaltsverzeichnis

1 Leitfaden zur Programmierung ... 1
2 Natural-Programmiermodi ... 5
  Sinn und Zweck ... 6
  Programmiermodus festlegen/ändern ... 7
  Funktionale Unterschiede ... 7
3 Objekttypen ... 15
4 Welche Typen von Programmierobjekten gibt es? ... 17
5 Datenbereiche (Data Areas) ... 19
  Verwendung von Datenbereichen ... 20
  Local Data Area ... 20
  Global Data Area ... 22
  Parameter Data Area ... 31
6 Programme, Functions, Subprogramme und Subroutinen ... 35
  Modulare Anwendungsstruktur ... 36
  Mehrere Stufen (Levels) aufgerufener Objekte ... 36
  Programm ... 38
  Function ... 41
  Subroutine ... 43
  Subprogramm ... 48
  Verarbeitungsablauf beim Aufruf eines Unterprogramms ... 50
7 Verarbeitung einer Rich GUI Page - Adapter ... 53
8 Map (Maske) ... 55
  Vorteile der Verwendung von Maps ... 56
  Map-Typen ... 56
  Maps erstellen ... 57
  Map-Verarbeitung starten/stoppen ... 57
9 Helproutinen ... 59
  Helproutinen aufrufen ... 60
  Helproutinen spezifizieren ... 60
  Programmierhinweise für Helproutinen ... 61
  Parameter an Helproutinen übergeben ... 61
  Gleichheitszeichen-Option ... 62
  Array-Felder ... 63
  Hilfe als eingeblendetes Fenster ... 63
10 Mehrfache Verwendung von Sourcecode — Copycode ... 65
  Copycode-Nutzung ... 66
  Copycode-Verarbeitung ... 66
11 Natural-Objekte dokumentieren — Text ... 67
  Verwendung des Natural-Objekttyps Text ... 68
  Text schreiben ... 68
12 Ereignisgesteuerte Anwendungen erstellen — Dialog ... 69
13 Komponentenbasierte Anwendungen erstellen — Class ... 71
14 Nicht-Natural-Dateien benutzen — Resource ... 73

Verwendung von Resource-Objekten ... 74
Shared Resources ... 74
Private Resources ... 75
15 Felder definieren ... 77
16 Benutzung und Struktur des DEFINE DATA-Statements ... 79
Felddefinitionen im DEFINE DATA-Statement ... 80
Felder innerhalb eines DEFINE DATA-Statements definieren ... 81
Felder in einer separaten Data Area definieren ... 81
Struktur eines DEFINE DATA-Statements — Level-Nummern ... 82
17 Benutzervariablen ... 85
Definition von Benutzervariablen ... 86
Datenbankfelder mit der (r)-Notation referenzieren ... 87
Sourcecode-Zeilennummern umnummerieren ... 88
Format und Länge von Benutzervariablen ... 89
Spezielle Formate ... 91
Index-Notation ... 93
Datenbank-Array referenzieren ... 97
Internen Zähler für ein Datenbank-Array referenzieren — C*-Notation ... 104
Datenstrukturen qualifizieren ... 108
Beispiele für Benutzervariablen ... 109
18 Function Call ... 111
Calling User-Defined Functions ... 112
Restrictions ... 113
Syntax Description ... 113
19 Dynamische Variablen ... 117
Sinn und Zweck dynamischer Variablen ... 118
Definition dynamischer Variablen ... 119
Zurzeit für eine dynamische Variable benutzter Wertespeicher ... 119
Größenbeschränkungsprüfung ... 120
Hauptspeicherplatz für eine dynamische Variable zuweisen/freigeben ... 120
20 Dynamische und große Variablen benutzen ... 123
Allgemeine Informationen zu dynamischen Variablen ... 124
Zuweisungen mit dynamischen Variablen ... 125
Initialisierung dynamischer Variablen ... 127
String-Manipulation mit dynamischen alphanumerischen Variablen ... 127
Logische Bedingungen (LCC) bei dynamischen Variablen ... 129
AT/IF-BREAK dynamischer Kontrollfelder ... 130
Parameter-Übertragung mit dynamischen Variablen ... 131
Arbeitsdateizugriff bei großen und dynamischen Variablen ... 134
DDM-Generierung und Editieren von Spalten mit variabler Länge ... 135
Zugriff auf große Datenbankobjekte ... 137
Performance-Aspekte bei dynamischen Variablen ... 138
Ausgabe von dynamische Variablen ... 140
Dynamische X-Arrays ... 140
21 Benutzerkonstanten ... 141

Numerische Konstanten ... 142
Alphanumerische Konstanten ... 143
Unicode-Konstanten ... 145
Datums- und Zeitkonstanten ... 148
Hexadezimale Konstanten ... 150
Logische Konstanten ... 152
Gleitkomma-Konstanten ... 152
Attribut-Konstanten ... 153
Handle-Konstanten ... 154
Namens-Konstanten definieren ... 154
22 Ausgangswerte (und das RESET-Statement) ... 155
Standard-Ausgangswert einer Benutzervariablen/ eines Arrays ... 156
Ausgangswert einer Benutzervariablen/einem Array zuweisen ... 156
Benutzervariable auf ihren Ausgangswert zurücksetzen ... 159
23 Felder redefinieren ... 161
REDEFINE-Option des DEFINE DATA-Statements ... 162
Beispielprogramm für eine Redefinition ... 164
24 Arrays ... 165
Arrays definieren ... 166
Ausgangswerte für Arrays ... 167
Ausgangswerte für eindimensionale Arrays zuweisen ... 167
Ausgangswerte für zweidimensionale Arrays zuweisen ... 168
Dreidimensionales Array ... 173
Arrays als Teil einer größeren Datenstruktur ... 174
Datenbank-Arrays ... 175
Arithmetische Ausdrücke in Index-Notationen ... 175
Arithmetische Funktionen bei Arrays ... 176
25 X-Arrays ... 179
Definition ... 180
Speicherverwaltung von X-Arrays ... 181
Speicherverwaltung von X-Gruppen-Arrays ... 181
X-Array referenzieren ... 183
Parameter-Übertragung mit X-Arrays ... 184
Parameter-Übertragung mit X-Group-Arrays ... 185
X-Array mit dynamischen Variablen ... 187
Unter- und Obergrenze eines Arrays ... 187
26 Benutzer-definierte Funktionen ... 189
Einführung in benutzer-definierte Funktionen ... 190
Unterschied zwischen Function Call und Subprogram Call ... 190
Definition einer Function (DEFINE FUNCTION) ... 192
Definition eines Prototype (DEFINE PROTOTYPE) ... 192
Symbolischer und variabler Function Call ... 193
Automatische/Implizite Prototype-Definition (APT) ... 193
Prototype Cast (PT-Klausel) ... 193
Zwischenergebnis für den Rückgabewert (IR-Klausel) ... 194

Kombinationen möglicher Prototype-Definitionen ... 194
Rekursiver Function Call ... 196
Behavior of Functions in Statements and Expressions ... 197
Usage of Functions as Statements ... 198
27 Datenbankzugriffe ... 201
28 Natural und Datenbankzugriff ... 203
Von Natural unterstützte Datenbankverwaltungssysteme ... 204
Profilparameter zur Beeinflussung der Datenbankzugriffe ... 205
Zugriff über Datendefinitionsmodule ... 205
Eingebaute Datenmanipulationssprache ... 206
Spezielle SQL-Statements in Natural ... 207
29 Daten in einer Adabas-Datenbank aufrufen ... 209
Adabas-Datenbankverwaltungsschnittstellen ADA und ADA2 ... 210
Datendefinitionsmodule (DDMs) ... 210
Datenbank-Arrays ... 212
Datenbank-View definieren ... 219
Statements für Datenbankzugriffe ... 222
Multi-Fetch-Klausel ... 235
Datenbank-Verarbeitungsschleifen ... 236
Datenänderungen - Transaktionsverarbeitung ... 243
Datensätze mit ACCEPT/REJECT auswählen ... 251
AT START/END OF DATA-Statements ... 255
Unicode-Daten ... 257
30 Daten in einer SQL-Datenbank aufrufen ... 259
Generating Natural DDMs ... 260
Setting Natural Profile Parameters ... 260
Natural DML Statements ... 261
Natural SQL Statements ... 268
Flexible SQL ... 276
RDBMS-Specific Requirements and Restrictions ... 277
Data-Type Conversion ... 280
Date/Time Conversion ... 280
Obtaining Diagnostic Information about Database Errors ... 282
SQL Authorization ... 282
31 Accessing Data in a Tamino Database ... 283
Prerequisite ... 284
DDM and View Definitions with Natural for Tamino ... 284
Natural Statements for Tamino Database Access ... 288
Natural for Tamino Restrictions ... 293
32 Steuerung der Ausgabe von Daten ... 295
33 Report-Spezifikation — (rep)-Notation ... 297
Report-Spezifikationen benutzen ... 298
Betroffene Statements ... 298
Beispiele für Report-Spezifikation ... 298
34 Layout einer Ausgabeseite ... 299

Statements mit Auswirkungen auf das Aussehen eines Report-Layouts 300
Allgemeines Layout-Beispiel 301
35 Statements DISPLAY und WRITE 303
Das DISPLAY-Statement 304
Das WRITE-Statement 305
Beispiel für ein DISPLAY-Statement 306
Beispiel für ein WRITE-Statement 307
Spaltenabstand - der SF-Parameter und die Notation nX 307
Tabulator-Notation nT 309
Zeilenvorschub — die Schrägstrich-Notation (/) 309
Weitere Beispiele für DISPLAY- und WRITE-Statements 312
36 Index-Notation für multiple Felder und Periodengruppen 313
Index-Notation benutzen 314
Beispiel für Index-Notation im DISPLAY-Statement 314
Beispiel für Index-Notation im WRITE-Statement 315
37 Seitenüberschriften, Seitenvorschübe und Leerzeilen 317
Standard-Seitenüberschrift 318
Seitenüberschrift unterdrücken — die NOTITLE-Option 318
Eigene Seitenüberschrift definieren — das WRITE TITLE-Statement 319
Logische Seite und physische Seite 323
Seitenlänge — der PS-Parameter 325
Seitenvorschub 325
Neue Seite mit Titel 328
Seiten-Fußzeile — das WRITE TRAILER-Statement 329
Leerzeilen erzeugen — das SKIP-Statement 331
AT TOP OF PAGE-Statement 333
AT END OF PAGE-Statement 334
Weiteres Beispiel 335
38 Spaltenüberschriften 337
Standard-Spaltenüberschriften 338
Standard-Spaltenüberschriften unterdrücken — die NOHDR-Option 339
Eigene Spaltenüberschriften definieren 339
NOTITLE und NOHDR kombinieren 340
Spaltenüberschriften zentrieren — der HC-Parameter 340
Breite von Spaltenüberschriften — der HW-Parameter 341
Füllzeichen für Überschriften — die Parameter FC und GC 341
Unterstreichungszeichen für Überschriften — der UC-Parameter 342
Spaltenüberschriften unterdrücken — die Schrägstrich-Notation ('/') 344
Weitere Beispiele für Spaltenüberschriften 345
39 Parameter zur Beeinflussung der Ausgabe von Feldern 347
Übersicht über Feldausgabe-relevante Parameter 348
Vorangestellte Zeichen — der LC-Parameter 348
Vorangestellte Zeichen im Unicode-Format — der LCU Parameter 349
Einfügungszeichen — der IC-Parameter 349
Einfügungszeichen im Unicode-Format — der ICU Parameter 350

Nachgestellte Zeichen — der TC-Parameter 350
Vorangestellte Zeichen im Unicode-Format — der TCU Parameter 350
Ausgabelänge — der AL- und der NL-Parameter 351
Ausgabelänge — der DL Parameter 351
Vorzeichen-Stelle — der SG-Parameter 353
Ausgabe identischer Werte unterdrücken — der IS-Parameter 355
Nullwerte anzeigen — der ZP-Parameter 357
Leerzeilen unterdrücken — der ES-Parameter 357
Weitere Beispiele für Feldausgabe-relevante Parameter 359
40 Codepage-Editiermasken — der EM-Parameter 361
Verwendung des EM-Parameters 362
Editiermasken für numerische Felder 363
Editiermasken für alphanumerische Felder 363
Länge der Felder 363
Editiermasken für Datums- und Zeitfelder 364
Trennzeichen-Angaben an lokale Standards anpassen 364
Beispiele für Editiermasken 366
Weitere Beispiele für Editiermasken 369
41 Unicode-Editiermasken — EMU-Parameter 371
42 Vertikale Ausgabe von Feldwerten 373
Vertikale Ausgaben erzeugen 374
Vertikale Ausgabe durch Kombination von DISPLAY und WRITE 374
Tabulator-Notation — T*field 375
Positionierungsnotation x/y 376
DISPLAY VERT-Statement 377
Weiteres Beispiel für DISPLAY VERT- mit WRITE-Statement 383
43 Weitere Programmieraspekte 385
44 Ende eines Statements, Programms oder einer Anwendung 387
Ende eines Statements 388
Ende eines Programms 388
Ende einer Anwendung 388
45 Verarbeitung von Anwendungsfehlern 391
Naturals Standard-Fehlerverarbeitung 392
Anwendungsspezifische Fehlerverarbeitung 393
Verwendung eines ON ERROR-Statement-Blocks 393
Verwendung eines Fehlertransaktionsprogramm 394
Funktionalität für die Fehlerverarbeitung 398
46 Bedingte Verarbeitung — Das IF-Statement 403
Struktur des IF-Statements 404
Geschachtelte IF-Statements 406
47 Schleifenverarbeitung 409
Verwendung von Verarbeitungsschleifen 410
Schleifendurchläufe bei Datenbankzugriffen begrenzen 410
Durchläufe bei Nicht-Datenbankzugriffsschleifen begrenzen — das REPEAT-Statement 412

Beispiel für Verarbeitungsschleife mit REPEAT-Statement 413
Verarbeitungsschleife verlassen — das ESCAPE-Statement 414
Schleifen innerhalb von Schleifen 414
Beispiel für geschachtelte FIND-Statements 415
Statements innerhalb eines Programms referenzieren 416
Beispiel für das Referenzieren mit Zeilennummern 418
Beispiel mit Statement-Labels 418
48 Gruppenwechsel 421
Verwendung von Gruppenwechseln 422
AT BREAK-Statement 422
Automatische Gruppenwechsel-Verarbeitung 428
Beispiel für Systemfunktionen in einem AT BREAK-Statement 430
Weiteres Beispiel für AT BREAK-Statement 431
BEFORE BREAK PROCESSING-Statement 431
Beispiel für BEFORE BREAK PROCESSING-Statement 431
Programmabhängige Gruppenwechsel-Verarbeitung — das PERFORM BREAK PROCESSING-Statement 432
Beispiel für PERFORM BREAK PROCESSING-Statement 434
49 Datenberechnungen 437
COMPUTE-Statement 438
Statements MOVE und COMPUTE 439
Statements ADD, SUBTRACT, MULTIPLY und DIVIDE 440
Beispiel für MOVE-, SUBTRACT- und COMPUTE-Statements 440
COMPRESS-Statement 441
Beispiel für die Statements COMPRESS und MOVE 442
Beispiel für COMPRESS-Statement 443
Mathematische Funktionen 444
Weitere Beispiele für COMPUTE-, MOVE- und COMPRESS-Statements 445
50 Systemvariablen und Systemfunktionen 447
Systemvariablen 448
Systemfunktionen 450
Beispiel für Systemvariablen und Systemfunktionen 450
Weitere Beispiele für Systemvariablen 452
Weitere Beispiele für Systemfunktionen 452
51 Natural-Stack 453
Verwendung des Natural-Stack 454
Stack-Verarbeitung 454
Daten im Stack ablegen 455
Stack-Inhalt löschen 456
52 Datumsinformationen verarbeiten 457
Editiermasken für Datumsfelder and Datumssystemvariablen 458
Standard-Editiermaske für Datum — der DTFORM-Parameter 458
Datumsformat für alphanumerische Darstellung — der DF-Parameter 459
Datumsformat für Ausgabe — der DFOUT-Parameter 462
Datumsformat für Stack — der DFSTACK-Parameter 463

Year Sliding Window — der YSLW-Parameter 464
Kombinationen von DFSTACK und YSLW 467
Year Fixed Window 469
Datumsformat für Standard-Seitenüberschriften — der DFTITLE-Parameter 470
53 Text-Notation 471
Mit einem Statement zu benutzenden Text definieren — die 'text'-Notation 472
Vor einem Feldwert n-mal anzuzeigendes Zeichen definieren — die 'c'(n)-Notation 474
54 Benutzerkommentare 475
Ganze Sourcecode-Zeile als Kommentarzeile benutzen 476
Teil einer Sourcecode-Zeile als Kommentarzeile benutzen 477
55 Logische Bedingungen 479
Einleitung 480
Relationaler Ausdruck 481
Erweiterter Relationaler Ausdruck 485
Auswertung einer logischen Variablen 486
Felder in logischen Bedingungen 488
Logische Operatoren in komplexen logischen Ausdrücken 490
BREAK-Option - Aktuellen Wert mit Wert des vorangegangenen Schleifendurchlaufs vergleichen 491
IS-Option - Prüfen ob Inhalt von Alphanumerischem oder Unicode-Feld konvertiert werden kann 493
MASK-Option - Ausgewählte Stellen eines Feldes auf bestimmten Inhalt prüfen 495
MASK-Option im Vergleich zur IS Option 504
MODIFIED-Option - Prüfen ob Feldinhalt verändert worden ist 506
SCAN-Option - Nach einem bestimmten Wert in einem Feld suchen 507
SPECIFIED-Option - Prüfen ob ein Wert für einen optionalen Parameter übergeben wird 509
56 Regeln für arithmetische Operationen 513
Initialisierung von Feldern 514
Kompatibilitätsregeln zur Datenübertragung 514
Abschneiden und Runden von Feldwerten 517
Format/Länge von Ergebnisfeldern bei arithmetischen Operationen 517
Arithmetische Operationen mit Gleitkomma-Zahlen 518
Arithmetische Operationen mit Datum und Zeit 520
Formatwahl im Hinblick auf die Verarbeitungszeit 524
Genauigkeit von Ergebnissen bei arithmetischen Operationen 525
Fehlerbedingungen bei arithmetischen Operationen 526
Verarbeitung von Arrays 526
57 Natural-Subprogramme aus 3GL-Programmen aufrufen 535
Parameter vom 3GL-Programm an das Subprogramm übergeben 536
Beispiel: Aufruf eines Natural-Subprogramms von einem 3GL-Programm 537
58 Betriebssystemkommandos aus einem Natural-Programm absetzen 539
Syntax 540

Parameter ... 540
Parameter-Optionen ... 540
Return-Codes ... 541
Beispiele ... 541
59 Statements für den Internet- und XML-Zugriff ... 543
Verfügbare Statements ... 544
Weitere Informationsquellen ... 545
60 Portierbare generierte Natural-Programme ... 547
Kompatibilität ... 548
Hinweise zum Endian-Mode ... 548
ENDIAN-Parameter ... 549
Natural-GPs übertragen ... 549
Portierbare FILEDIR.SAG- und Fehlermeldungsdateien ... 551
61 Introduction to Event-Driven Programming ... 553
62 What is an Event-Driven Application? ... 555
Introduction ... 556
Program-Driven Applications ... 557
Event Driven Applications ... 558
What is Happening Here? ... 559
Writing Event-Driven Code ... 559
Components of an Event Driven Application ... 560
63 GUI Development Environments ... 563
64 GUI Design Tips ... 565
Introduction ... 566
Do Your Research ... 566
Screen Design ... 567
Menu Design ... 568
Color Usage ... 569
Consistency Check ... 569
65 Tasks Involved in Creating an Application ... 571
66 Tutorial ... 573
Creating a Dialog ... 574
Assigning Attributes to the Dialog ... 575
Creating Dialog Elements Inside the Dialog ... 577
Assigning Attributes to the Dialog Elements ... 579
Creating the Application's Local Data Area ... 580
Attaching Event Handler Code to the Dialog Element ... 581
Checking, Stowing and Running the Application ... 582
67 Basic Terminology ... 585
Attribute ... 586
Base Dialog ... 586
Control ... 587
Dialog ... 587
Dialog Box ... 587
Dialog Editor ... 587

Dialog Element ... 587
Event ... 588
Event Handler ... 588
Handle ... 588
Item ... 588
MDI - Multiple Document Interface ... 588
MDI Child Window ... 589
MDI Frame Window ... 589
Modal Window ... 589
SDI - Single Document Interface ... 589
Popup ... 589
Window ... 589
68 Event-Driven Programming Techniques ... 591
69 Introduction ... 595
70 How To Open and Close Dialogs ... 597
Opening a Dialog ... 598
Operands ... 598
Passing Parameters to the Dialog ... 599
Permanence in Creating, Passing and Checking Data ... 600
Processing Steps When Opening a Dialog ... 601
Closing Dialogs ... 602
Initializing Attribute Values ... 602
71 How To Edit a Dialog's Enhanced Source Code ... 605
What Is The Enhanced Source Code Format? ... 606
Avoiding Incompatibilities Between Dialog Editor And Program Editor ... 607
How To Use The Enhanced Source Code Format ... 608
72 How Dialogs, Controls and Items Are Related Hierarchically ... 609
73 How To Define Dialog Elements ... 611
Introduction ... 612
HANDLE OF GUI ... 613
NULL-HANDLE ... 613
74 How To Manipulate Dialog Elements ... 615
Introduction ... 616
Querying, Setting and Modifying Attribute Values ... 616
Restrictions ... 617
Numeric/Alphanumeric Assignment ... 618
75 How To Create and Delete Dialog Elements Dynamically ... 619
Introduction ... 620
Global Attribute List ... 620
Creating Dialog Elements Statically and Dynamically ... 620
How to Handle Events of Dynamically Created Dialog Elements ... 622
76 How To Enable and Disable Dialog Elements ... 625
77 Defining and Using Context Menus ... 627
Introduction ... 628
Construction ... 628

- Association ... 629
- Invocation ... 630
- Manual Invocation ... 633
- Sharing of Context Menus ... 635

78 Using the Clipboard and Drag and Drop ... 637
- Introduction ... 638
- Clipboard Specifics ... 640
- Drag and Drop Specifics ... 641
- Drag and Drop Insertion Marks ... 643
- Drag-Drop Checklist ... 644

79 System Variables ... 647

80 Generated Variables ... 649
- #DLG$PARENT ... 650
- #DLG$WINDOW ... 650

81 Message Files and Variables as Sources of Attribute Values ... 651

82 Triggering User-Defined Events ... 653
- Introduction ... 654
- Passing Parameters to the Dialog ... 655

83 Suppressing Events ... 657

84 Menu Structures, Toolbars and the MDI ... 659
- Creating a Menu Structure ... 660
- Parent-Child Hierarchy in Menu Structures ... 662
- Creating a Toolbar ... 662
- Sharing Menu Structures, Toolbars and DILs (MDI Application) ... 663

85 Executing Standardized Procedures ... 665
- Introduction ... 666
- PROCESS GUI Statement ... 666

86 Linking Dialog Elements to Natural Variables ... 667

87 Validating Input in a Dialog Element ... 669

88 Storing and Retrieving Client Data for a Dialog Element ... 671
- Introduction ... 672
- Integer Data ... 672
- Handle Data ... 673
- Keyed Alphanumeric Client Data ... 673
- Keyed Client Data in Native Format ... 676
- Key Enumeration ... 679

89 Creating Dialog Elements on a Canvas Control ... 681

90 Label Editing in Tree View and List View Controls ... 685
- Introduction ... 686
- Label Editing ... 686
- Changing an Item's Label Programmatically ... 688

91 Working with ActiveX Controls ... 689
- Terminology ... 690
- How To Define an ActiveX Control ... 690
- How To Create an ActiveX Control ... 690

Accessing Simple Properties 691
Colors 693
Pictures 693
Fonts 694
Variants 695
Arrays 696
Using the PROCESS GUI Statement 697
92 Working with Arrays of Dialog Elements 703
93 Working with Control Boxes 705
Introduction 706
Purpose of Exclusive Control Boxes 706
Examples of Use of Exclusive Control Boxes 707
Creation of the Wizard Pages 708
94 Working with Date and Time Picker (DTP) Controls 713
Introduction 714
Date and Time Formats 714
Inputting Dates and Times 715
Null Values 716
Calendar Colors and Font 716
95 Working with Dialog Bar Controls 717
Introduction 718
Creating a Dialog Bar Control 718
Types of Dialog Bar Control 718
UI Transparency 722
Client-Size Event 722
Close Button 722
Sample Code 722
96 Working with Error Events 727
97 Working with a Group of Radio Button Controls 729
98 Working with Image List Controls 731
Introduction 732
Creating the Image List Control 732
Adding Images 732
Composite Images 733
Scaling and Transparency 734
Bitmaps vs. Icons 735
Using an Image List 736
Referencing Images from the Image List 736
Overlay Images 737
Modifying Images 738
Deleting Images 739
Deleting the Image List Control 739
99 Working with List Box Controls and Selection Box Controls 741
100 Working with List View Controls 745
Introduction 746

View Modes 746
Setting Item Images 748
Item Placement 748
Item Selection 750
Item Activation 752
List View Columns and Sub-items 752
Sorting 755
Label Editing 757
Multiple Context Menus 759
Drag and Drop 760
101 Working with Nested Controls 765
Introduction 766
Which Control Types can be Containers? 767
Creating a Nested Control 767
Multiple Selection, Control Sequence and Clipboard Operations 768
102 Working with a Dynamic Information Line 771
103 Working with Spin Controls 773
Introduction 774
Up-Down Control 774
Buddy Control 774
Date and Time Formats 775
Inputting Dates and Times 776
Null Values 777
Calendar Colors and Font 777
104 Working with a Status Bar 779
105 Working with Status Bar Controls 781
Introduction 782
Creating a Status Bar Control 782
Using Status Bar Controls without Panes 782
Outputting Text to a Status Bar Control 783
Sharing a Status Bar in an MDI Application 784
Pane-specific Context Menus 785
106 Working with Tab Controls 787
Creating a Tab Control 788
Assigning Controls to Tabs 788
Use of Control Boxes as Tab Control Pages 789
Switching Between Controls Belonging To Different Tabs 790
Mixing Tab-dependent and Tab-independent Controls 791
Keyboard Navigation 791
Tab Switching Events 792
107 Working with Tree View Controls 793
Introduction 794
Setting Item Images 794
Item Selection 795
Item Activation 795

Item Data 796
Sorting 797
Label Editing 797
Multiple Context Menus 798
Dynamic Item Creation 799
Drag and Drop 800
108 Working with Dynamic Information Line and Status Bar 805
109 Adding a Maximize/Minimize/System Button 807
110 Defining Color 809
111 Adding Text in a Certain Font 811
112 Adding Online Help 813
113 Defining Mnemonic and Accelerator Keys 817
Introduction 818
Defining a Mnemonic Key 818
Defining an Accelerator Key 819
Displaying Accelerator Keys in Menus 819
114 Dynamic Data Exchange - DDE 821
Concepts 822
Developing a DDE Server Application 823
Developing a DDE Client Application 824
Return Codes 825
115 Object Linking and Embedding - OLE 829
What is OLE in the Natural Context? 830
OLE Documents Support 830
Embedding and Linking 830
Visual Editing - In-place Activation 831
ActiveX Controls Support 832
OLE Container Control 832
Attributes, Events and PROCESS GUI Statement Actions 835
116 Results Interface 837
Zweck des Results Interfaces 838
Auf das Results-Fenster zugreifen 839
Registerkarten 839
Bilder 840
Kontextmenüs 840
Befehle 841
Spalten 841
Zeilen 842
Daten 842
Markierungen 842
117 Gestaltung von zeichenbasierten Benutzeroberflächen von Anwendungen 843
118 Bildschirmgestaltung 845
Steuerung der Meldungszeile — Terminalkommando %M 846
Zuweisen von Farben zu Feldern — Terminalkommando %= 846
Statistikzeile/Infoline — Terminalkommando %X 848

Fenster ... 848
Standard-/Dynamische Layout-Maps ... 854
Mehrsprachige Benutzeroberflächen ... 855
Kenntnisabhängige Benutzeroberflächen (Expertenmodus) ... 860
119 Dialog-Gestaltung ... 863
Feldabhängige Verarbeitung ... 864
Einfachere Programmierung ... 866
Zeilenabhängige Verarbeitung ... 867
Spaltenabhängige Verarbeitung ... 868
Verarbeitung aufgrund von Funktionstasten ... 869
Verarbeitung aufgrund der Namen von Funktionstasten ... 870
Verarbeitung von Daten außerhalb des aktiven Fensters ... 870
Daten vom Bildschirm kopieren ... 874
Statements REINPUT/REINPUT FULL ... 877
Objektorientierte Datenverarbeitung — der Natural-Kommando-Prozessor ... 878
120 Natural Native Interface ... 879
121 Introduction ... 881
122 Interface Library and Location ... 883
123 Interface Versioning ... 885
124 Interface Access ... 887
125 Interface Instances and Natural Sessions ... 889
126 Interface Functions ... 891
nni_get_interface ... 893
nni_free_interface ... 894
nni_initialize ... 894
nni_is_initialized ... 896
nni_uninitialize ... 897
nni_enter ... 897
nni_try_enter ... 898
nni_leave ... 898
nni_logon ... 899
nni_logoff ... 900
nni_callnat ... 901
nni_function ... 902
nni_create_object ... 903
nni_send_method ... 904
nni_get_property ... 906
nni_set_property ... 907
nni_delete_object ... 909
nni_create_parm ... 910
nni_create_module_parm ... 911
nni_create_method_parm ... 912
nni_create_prop_parm ... 913
nni_parm_count ... 914
nni_init_parm_s ... 915

nni_init_parm_sa 916
nni_init_parm_d 917
nni_init_parm_da 918
nni_get_parm_info 919
nni_get_parm 920
nni_get_parm_array 921
nni_get_parm_array_length 922
nni_put_parm 923
nni_put_parm_array 924
nni_resize_parm_array 925
nni_delete_parm 926
nni_from_string 927
nni_to_string 928
127 Parameter Description Structure 931
128 Natural Data Types 933
129 Flags 935
130 Return Codes 937
131 Natural Exception Structure 939
132 Interface Usage 941
133 Threading Issues 943
134 NaturalX 945
135 Einführung in NaturalX 947
Warum NaturalX? 948
Programming Techniques 949
136 NaturalX-Anwendungen entwickeln 953
Entwickungsumgebungen 954
Klassen definieren 954
Klassen und Objekte benutzen 959
137 Distributing NaturalX Applications 965
General 966
Globally Unique Identifiers - GUIDs 968
138 ActiveX Component SoftwareAG.NaturalX.Utilities 969
Purpose 970
Interfaces 970
139 Interface INaturalXUtilities 971
Purpose 972
Methods 972
140 Interface IRunningObjects 975
Purpose 976
Methods 978
141 ActiveX Component SoftwareAG.NaturalX.Enumerator 979
Purpose 980
Interface 981
142 Interface IEnumerator 983
Purpose 984

Methods ... 984
143 Für Natural reservierte Schlüsselwörter ... 987
Alphabetische Liste der für Natural reservierten Schlüsselwörter ... 988
Prüfung auf für Natural reservierte Schlüsselwörter durchführen ... 1003
144 Referenzierte Beispielprogramme ... 1007
READ-Statement ... 1008
FIND-Statement ... 1009
Geschachtelte READ- und FIND-Statements ... 1013
ACCEPT- und REJECT-Statements ... 1015
AT START OF DATA- und AT END OF DATA-Statements ... 1017
DISPLAY- und WRITE-Statements ... 1020
DISPLAY-Statement ... 1024
Spaltenüberschriften ... 1025
Feldausgabe-relevante Parameter ... 1027
Editiermasken ... 1033
DISPLAY VERT mit WRITE-Statement ... 1036
AT BREAK-Statement ... 1037
Statements COMPUTE, MOVE und COMPRESS ... 1038
Systemvariablen ... 1041
Systemfunktionen ... 1044

1 Leitfaden zur Programmierung

Dieses Dokument ergänzt die Natural-Referenzdokumentation und liefert grundlegende Informationen zu verschiedenen Aspekten der Programmierung mit Natural. Sie sollten zuerst mit diesen Informationen vertraut sein, bevor Sie anfangen, Natural-Anwendungen zu schreiben.

Siehe auch *Erste Schritte*. Dieses Tutorial enthält eine Reihe von kurzen Anleitungen, die Sie in einige der Grundlagen der Natural-Programmierung einführen.

	Natural-Programmiermodi	Beschreibt die Unterschiede zwischen den beiden Natural-Programmiermodi: Reporting Mode und Structured Mode. Grundsätzlich empfiehlt es sich, ausschließlich im Structured Mode zu programmieren, um eine klar strukturierte Anwendung zu gewährleisten. Daher beziehen sich alle Erklärungen und Beispiele in dieser Dokumentation auf den Structured Mode. Besonderheiten des Reporting Mode werden in der Regel berücksichtigt.
	Objekttypen	Innerhalb einer Anwendung können Sie verschiedene Typen von Programmierobjekten benutzen, um eine effiziente Anwendungsstruktur zu erhalten. Dieses Dokument beschreibt die verschiedenen Typen von Natural-Programmierobjekten: Datenbereiche (Data Areas), Programme, Subprogramme, Subroutinen, Helproutinen, Maps usw.
	Felder definieren	Beschreibt, wie Sie die Felder definieren, die Sie in einem Programm verwenden möchten.
	Benutzer-definierte Funktionen	Beschreibt die Vorteile der Verwendung des Programmierobjekts „Function", erläutert die Unterschiede bei der Verwendung von „Function Calls" und „Subprogram Calls" und stellt die verfügbaren Methoden zum Definieren und Aufrufen einer „Function" vor.
	Datenbankzugriffe	Beschreibt, wie Sie mit Natural auf Daten in einer Adabas-Datenbank und auf Daten in verschiedenen, von Natural unterstützten Nicht-Adabas-Datenbanken zugreifen können. Im Prinzip gelten die für Adabas beschriebenen Funktionen und Beispiele auch für andere Datenbankverwaltungssysteme. Abweichungen, falls

		vorhanden, sind in der betreffenden Schnittstellen-Dokumentation und in der *Statements*-Dokumentation oder der *Parameter-Referenz* beschrieben.
●	**Steuerung der Ausgabe von Daten**	Beschreibt, wie Sie das Aussehen eines mit Natural erzeugten Ausgabe-Reports, d.h. die Art, in der die Daten angezeigt werden, beeinflussen können.
●	**Weitere Programmieraspekte**	Beschreibt verschiedene andere Aspekte der Programmierung mit Natural.
●	**Portierbare generierte Natural-Programme**	Ab Natural Version 5 sind generierte Programme (GPs) auf den von Natural unterstützten UNIX-, OpenVMS- und Windows-Plattformen portierbar.
●	**Introduction to Event-Driven Programming**	Enthält grundsätzliche Informationen zur ereignisgesteuerten Programmierung. (Dieser Teil ist nur in Englisch verfügbar.)
●	**Event-Driven Programming Techniques**	Enthält Informationen für erfahrene GUI-Programmierer und beschreibt die Programmiertechniken. (Dieser Teil ist nur in Englisch verfügbar.)
●	**Results Interface**	Mit dem Results Interface können Daten im Results-Fenster (Ergebnisfenster) von Natural Studio angezeigt werden.
●	**Gestaltung von zeichenbasierten Benutzeroberflächen von Anwendungen**	Enthält Informationen zu Bestandteilen von Natural, die Sie zur Gestaltung von zeichenbasierten Benutzeroberflächen Ihrer Anwendungen benutzen können.
●	**Natural Native Interface**	Beschreibt das Natural Native Interface, durch das Nicht-Natural-Anwendungen in die Lage versetzt werden, Natural-Code unter Verwendung von C-Funktionsaufrufen auszuführen. (Dieser Teil ist nur in Englisch verfügbar.)
●	**NaturalX**	Beschreibt, wie Sie objekt-basierte Anwendungen entwickeln und verteilen.
●	**Für Natural reservierte Schlüsselwörter**	Enthält eine Liste aller Natural-Schlüsselwörter und für Natural reservierten Wörter.
●	**Referenzierte Beispielprogramme**	Der *Leitfaden zur Programmierung* enthält zahlreiche Beispiele für Natural-Programme sowie Verweise auf weitere Beispielprogramme (hauptsächlich für Reporting Mode), die separat in diesem Kapitel zusammengefasst sind. **Anmerkung zu Beispiel-Libraries:** ■ Den Sourcecode all dieser Beispielprogramme finden Sie in der Natural-Library `SYSEXPG`. Die Beispielprogramme greifen auf die Daten der Dateien `EMPLOYEES` (Personaldaten) und `VEHICLES` (Fahrzeugdaten) zu, die von der Software AG speziell zu Demonstrationszwecken erstellt wurden. ■ Weitere Beispielprogramme für die Verwendung von Natural-Statements finden Sie in der Natural-Library `SYSEXSYN`. Diese Beispiele sind außerdem im Abschnitt *Referenzierte Beispielprogramme* in der *Statements*-Dokumentation enthalten.

		■ Wenden Sie sich wegen der Verfügbarkeit dieser Bibliotheken in Ihrem Unternehmen an Ihren Natural-Administrator. ■ Damit die Natural-Beispielprogramme auf eine Adabas-Datenbank zugreifen können, muss der Adabas-Nucleus-Parameter `OPTIONS` auf `TRUNCATION` gesetzt sein.

2 Natural-Programmiermodi

- Sinn und Zweck 6
- Programmiermodus festlegen/ändern 7
- Funktionale Unterschiede 7

Dieses Kapitel beschreibt die zwei von Natural unterstützten Programmiermodi.

Sinn und Zweck

Natural bietet zwei Formen der Programmierung:

- Reporting Mode
- Structured Mode

Anmerkung: Grundsätzlich empfiehlt es sich, ausschließlich im Structured Mode zu arbeiten, um klar strukturierte Anwendungen zu erhalten.

Reporting Mode

Der Reporting Mode eignet sich nur für die Erstellung einfacher Reports und Programme, die keine komplexe Daten- und Programmstruktur erfordern. (Falls Sie sich entschließen sollten, ein Programm im Reporting Mode zu schreiben, sollten Sie bedenken, dass kleine Programme schnell umfangreicher und komplexer werden können.)

Bitte achten Sie darauf, dass manche Natural-Statements nur im Reporting Mode verfügbar sind, wohingegen andere eine spezifische Struktur aufweisen, wenn Sie im Reporting Mode benutzt werden. Eine Übersicht der Statements, die im Reporting Mode verwendet werden können, entnehmen Sie dem Abschnitt *Statements im Reporting Mode* in der *Statements*-Dokumentation.

Structured Mode

Der Structured Mode ist für komplexe Anwendungen gedacht, bei denen es auf eine klare und sinnvoll gegliederte Programmstruktur ankommt. Wesentliche Vorteile des Structured Mode sind:

- Die Programme müssen strukturierter geschrieben werden und sind daher leichter zu lesen und folglich auch leichter zu pflegen.
- Da alle in einem Programm verwendeten Felder an einer zentralen Stelle definiert werden müssen (und nicht, wie im Reporting Mode, über das ganze Programm verstreut sein dürfen), wird der Überblick über die verwendeten Daten erheblich erleichtert.

Darüber hinaus zwingt der Structured Mode zu einer genaueren Anwendungsplanung, bevor es an das eigentliche Programmieren geht. Viele Fehler und Unzulänglichkeiten bei der Programmierung werden dadurch von vorneherein vermieden.

Eine Übersicht der im Structured Mode zu benutzenden Statements entnehmen Sie dem Abschnitt *Statements nach Funktionsgruppen* in der *Statements*-Dokumentation.

Programmiermodus festlegen/ändern

Der Standardprogrammiermodus wird vom Natural-Administrator mit dem Profilparameter `SM` festgelegt.

Weitere Informationen zum Profil- und Session-Parameter `SM`, entnehmen Sie dem Abschnitt *SM - Programming in Structured Mode* in der *Parameter-Referenz*-Dokumentation.

Sie können den vorgegebenen Modus mit dem Systemkommando `GLOBALS` und dem Session-Parameter `SM` ändern:

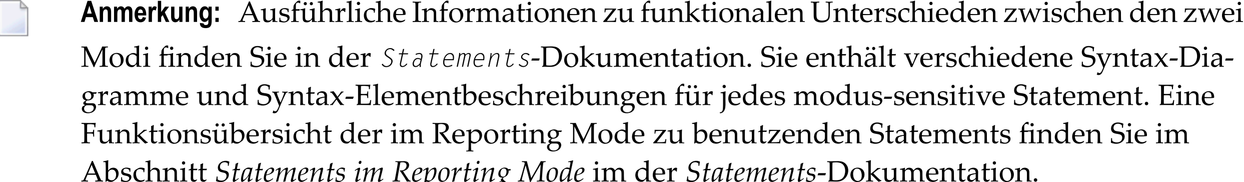
Structured Mode:	`GLOBALS SM=ON`
Reporting Mode:	`GLOBALS SM=OFF`

Weitere Informationen, wie Sie den Programmiermodus ändern können, finden Sie in folgenden Dokumenten: *SM - Programming in Structured Mode* in der *Parameter-Referenz*-Dokumentation.

Funktionale Unterschiede

Die wichtigsten funktionalen Unterschiede zwischen Reporting Mode und Structured Mode lassen sich wie folgt zusammenfassen:

- Syntax zum Beenden von Schleifen und funktionalen Blöcken
- Verarbeitungsschleife im Reporting Mode beenden
- Verarbeitungsschleife im Structured Mode beenden
- Platzierung von Datenelementen in einem Programm
- Datenbank-Referenzierung



Anmerkung: Ausführliche Informationen zu funktionalen Unterschieden zwischen den zwei Modi finden Sie in der `Statements`-Dokumentation. Sie enthält verschiedene Syntax-Diagramme und Syntax-Elementbeschreibungen für jedes modus-sensitive Statement. Eine Funktionsübersicht der im Reporting Mode zu benutzenden Statements finden Sie im Abschnitt *Statements im Reporting Mode* im der *Statements*-Dokumentation.

Syntax zum Beenden von Schleifen und funktionalen Blöcken

Reporting Mode:	(CLOSE) LOOP und DO ... DOEND-Statements werden hierzu verwendet. END-...-Statements (außer END-DEFINE, END-DECIDE und END-SUBROUTINE) können nicht benutzt werden.
Structured Mode:	Jede Schleife oder logische Verarbeitungsbedingung muss ausdrücklich mit einem entsprechenden END-...-Statement abgeschlossen werden. Dadurch wird sofort deutlich, wo welche Schleife bzw. Bedingung aufhört. LOOP und DO/DOEND-Statements können nicht benutzt werden.

Die beiden folgenden Beispiele veranschaulichen die je nach Programmiermodus unterschiedliche Konstruktion von Verarbeitungsschleifen und logischen Bedingungen.

Beispiel — Reporting Mode:

Im Reporting-Mode-Beispiel werden die Statements DO und DOEND verwendet, um den Statement-Block einzugrenzen, der an die AT END OF DATA-Bedingung geknüpft ist.

```
READ EMPLOYEES BY PERSONNEL-ID
DISPLAY NAME BIRTH
AT END OF DATA
   DO
      SKIP 2
      WRITE / 'LAST SELECTED:' OLD(NAME)
   DOEND
END
```

Das END-Statement beendet sämtliche aktiven Verarbeitungsschleifen.

Beispiel — Structured Mode:

Im Structured-Mode-Beispiel wird ein END-ENDDATA-Statement verwendet, um die AT END OF DATA-Bedingung zu beenden, sowie ein END-READ-Statement, um die READ-Schleife zu beenden. Das Ergebnis ist ein deutlicher strukturiertes Programm, in dem Sie sofort sehen können, wo welche Konstruktion anfängt und aufhört:

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH

END-DEFINE
READ MYVIEW BY PERSONNEL-ID
```

```
   DISPLAY NAME BIRTH
   AT END OF DATA
      SKIP 2
      WRITE / 'LAST SELECTED:' OLD(NAME)
   END-ENDDATA
END-READ
END
```

Verarbeitungsschleife im Reporting Mode beenden

Zum Beenden einer Verarbeitungsschleife können Sie im Reporting Mode die Statements `END`, `LOOP` (bzw. `CLOSE LOOP`) oder `SORT` verwenden.

Mit dem `LOOP`-Statement können Sie mehrere Schleifen gleichzeitig schließen. Mit dem `END`-Statement können Sie sämtliche noch nicht beendeten Schleifen schließen. Diese Möglichkeit, mehrere Schleifen mit einem einzigen Statement zu beenden, stellt einen grundlegenden Unterschied zum Structured Mode dar.

Ein `SORT`-Statement beendet alle Schleifen und initiiert gleichzeitig eine neue Schleife.

Beispiel 1 — LOOP:

```
FIND ...
   FIND ...
   ...
   ...
   LOOP      /* closes inner FIND loop
LOOP         /* closes outer FIND loop
...
...
```

Beispiel 2 — END:

```
FIND ...
   FIND ...
   ...
   ...
END             /* closes all loops and ends processing
```

Beispiel 3 — SORT:

```
FIND ...
   FIND ...
   ...
   ...
SORT ...        /* closes all loops, initiates loop
...
END             /* closes SORT loop and ends processing
```

Verarbeitungsschleife im Structured Mode beenden

Im Structured Mode gibt es zum Beenden jeder Verarbeitungsschleife ein bestimmtes Statement. Mit dem `END`-Statement werden keine Schleifen geschlossen. Bei Verwendung des `SORT`-Statements müssen Sie vorher ein `END-ALL`-Statement verwenden, sowie zum Beenden der `SORT`-Schleife ein `END-SORT`-Statement.

Beispiel 1 — FIND:

```
FIND ...
   FIND ...
   ...
   ...
   END-FIND       /* closes inner FIND loop
END-FIND          /* closes outer FIND loop
...
```

Beispiel 2 — READ:

```
READ ...
   AT END OF DATA
   ...
   END-ENDDATA
   ...
END-READ           /* closes READ loop
...
...
END
```

Beispiel 3 — SORT:

```
READ ...
   FIND ...
   ...
   ...
END-ALL         /* closes all loops
SORT            /* opens loop
...
...
END-SORT        /* closes SORT loop
END
```

Platzierung von Datenelementen in einem Programm

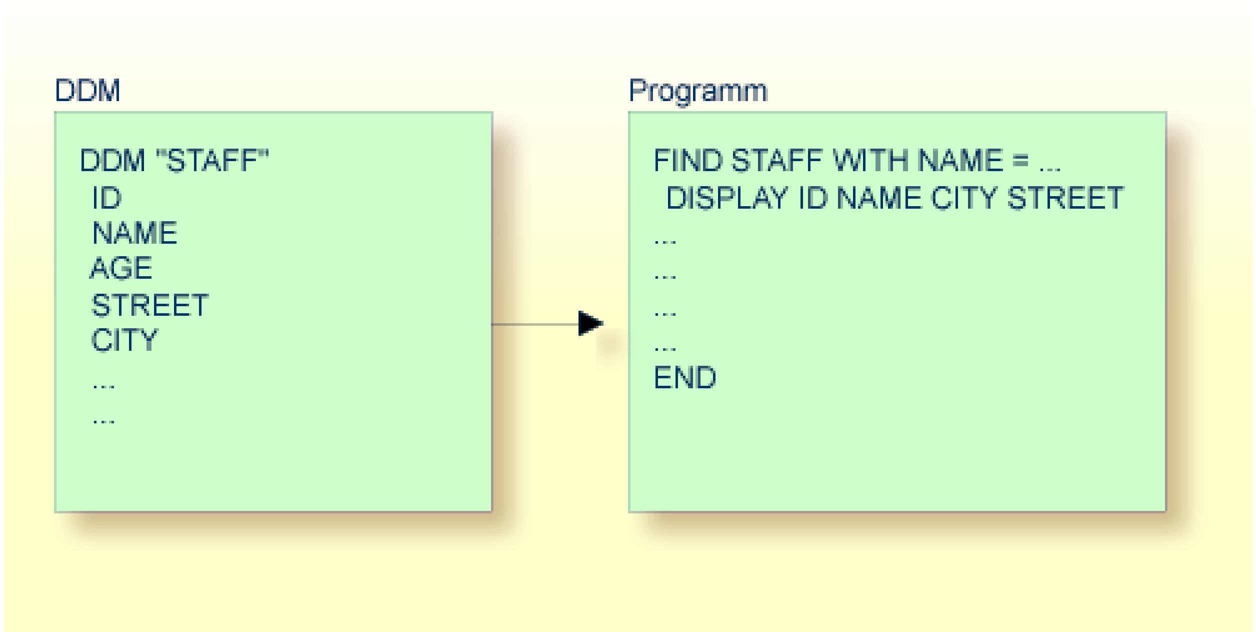
Im Reporting Mode können Datenbankfelder benutzt werden, ohne dass diese vorher in einem `DEFINE DATA`-Statement definiert werden müssen; außerdem können Benutzervariablen an jeder Stelle eines Programms, d.h. über das ganze Programm verstreut, definiert werden.

Im Structured Mode dagegen müssen *alle* verwendeten Datenelemente an einer zentralen Stelle (entweder im `DEFINE DATA`-Statement am Anfang des Programms oder in einer externen **Data Area**) definiert werden.

Datenbank-Referenzierung

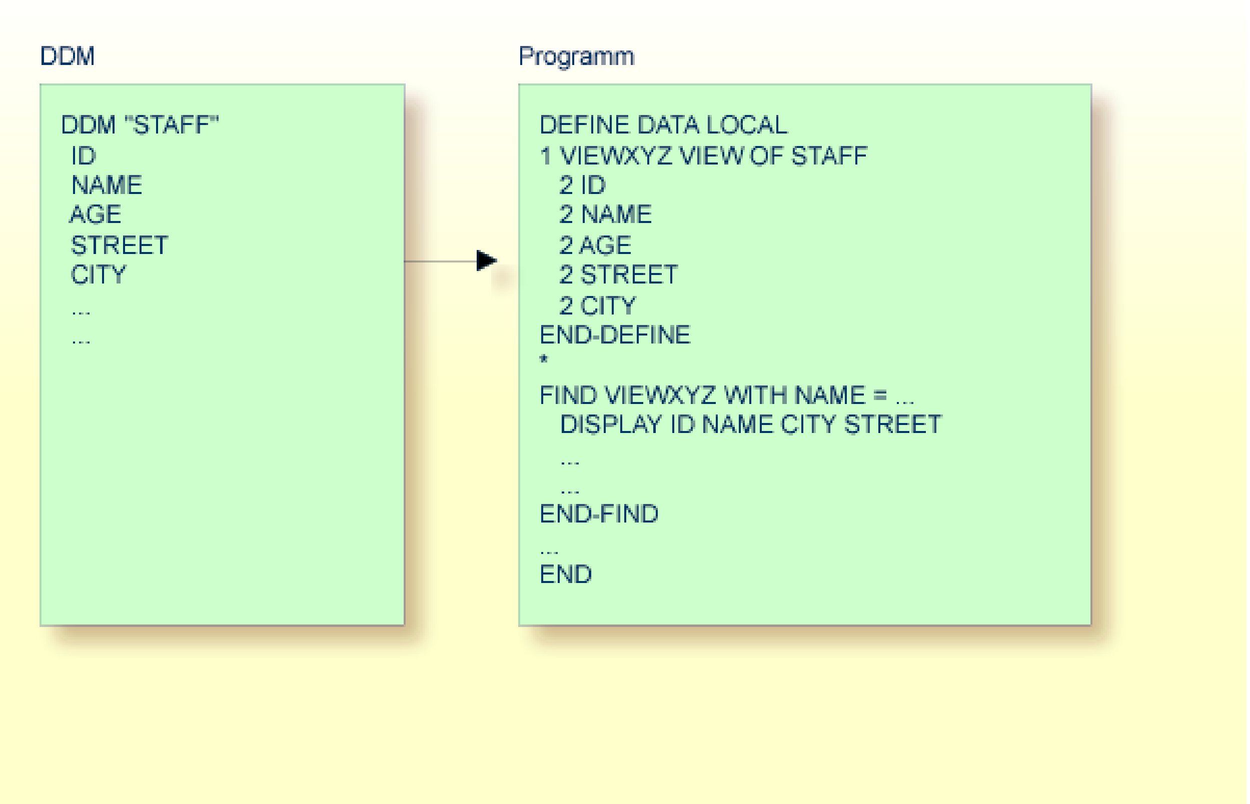
Reporting Mode:

Im Reporting Mode ist es möglich, Datenbankfelder oder DDMs zu benutzen, ohne diese in einer **Data Area** definiert zu haben.



Structured Mode:

Im Structured Mode dagegen muss jedes Datenbankfeld, das benutzt werden soll, in einem `DEFINE DATA`-Statement angegeben werden (wie in den Abschnitten ***Felder definieren*** und ***Datenbankzugriffe*** beschrieben).


DDM
Programm
DDM "STAFF"
ID
NAME
AGE
STREET
CITY
...
...
DEFINE DATA LOCAL
1 VIEWXYZ VIEW OF STAFF
2 ID
2 NAME
2 AGE
2 STREET
2 CITY
END-DEFINE
*
FIND VIEWXYZ WITH NAME = ...
DISPLAY ID NAME CITY STREET
...
...
END-FIND
...
END

3 Objekttypen

Dieser Teil beschreibt verschiedene Natural-Objekttypen, die zur Gewährleistung einer effizienten Anwendungsstruktur benutzt werden können. Alle Natural-Objekte werden in Natural-Libraries gespeichert. Natural-Libraries sind in Natural-Systemdateien enthalten.

Folgende Themen werden behandelt:

- Welche Typen von Programmierobjekten gibt es?
- Datenbereiche (Data Areas)
- Programme, Functions, Subprogramme und Subroutinen
- Verarbeitung einer Rich GUI Page - Adapter
- Map (Maske)
- Helproutinen
- Mehrfache Verwendung von Sourcecode - Copycode
- Natural-Objekte dokumentieren - Text
- Ereignisgesteuerte Anwendungen erstellen - Dialog
- Komponentenbasierte Anwendungen erstellen - Class
- Nicht-Natural-Dateien benutzen - Resource

4 Welche Typen von Programmierobjekten gibt es?

Dieses Kapitel enthält eine Übersicht über die Typen von Programmierobjekten, die innerhalb einer Natural-Anwendung verwendet werden können und benennt zu jedem Typ den passenden Natural-Editor.

All diese Objekte erstellen und pflegen Sie mit den Natural-Editoren.

Objekttyp	Natural-Editor
Programm	
Class	
Subprogramm	
Function *	
Adapter	
Subroutine	
Copycode	
Helproutine	
Text	
Class	Class Builder
Dialog **	Dialog Editor
Map	Map-Editor
Local Data Area	Data-Area-Editor
Global Data Area	
Parameter Data Area	
Resource ***	Externes Objekt, kann nicht mit Natural-Editoren bearbeitet werden.

** Der Objekttyp Dialog und der Dialog-Editor sind nur bei Natural for Windows verfügbar.

*** Der Objekttyp Resource ist auf Großrechnern nicht verfügbar.

Siehe auch *Natural-Objekte verwalten* in der Dokumentation *Natural Studio benutzen*.

5 Datenbereiche (Data Areas)

- Verwendung von Datenbereichen 20
- Local Data Area 20
- Global Data Area 22
- Parameter Data Area 31

Ein Natural-Datenbereich (Data Area) ist ein Natural-Programmierobjekt, das von mehreren Natural-Programmen, -Subprogrammen, -Subroutinen, Helproutinen, Dialoge oder Klassen benutzt werden kann.

Verwendung von Datenbereichen

Wie im Abschnitt ***Felder definieren*** erläutert, müssen alle Felder, die in einem Programm verwendet werden sollen, mit einem `DEFINE DATA`-Statement definiert werden.

Die Felder können entweder innerhalb des `DEFINE DATA`-Statements selbst definiert werden oder sie können außerhalb des Programms in einem separaten Datenbereich (Data Area) definiert werden, der dann vom `DEFINE DATA`-Statement referenziert wird.

Eine solche separate Data Area ist ein Natural-Programmierobjekt, das von mehreren Natural-Programmen, -Subprogrammen, -Subroutinen, Helproutinen, Dialoge oder Klassen benutzt werden kann. Eine Data Area enthält Datenelement-Definitionen, wie z.B. benutzerdefinierte Variablen, Konstanten und Datenbankfelder aus einem Datendefinitionsmodul (DDM).

Alle Data Areas werden mit dem Data Area Editor erstellt und editiert.

Mit Natural können Sie folgende Arten von Data Areas anlegen und referenzieren:

- **Local Data Area**
- **Global Data Area**
- **Parameter Data Area**

Local Data Area

Als „local“ definierte Variablen können nur von einem einzigen Natural-Programmierobjekt benutzt werden.

Sie haben zwei Möglichkeiten, lokale Daten zu definieren:

- Sie können die Daten innerhalb des Programms definieren.
- Sie können die Daten außerhalb des Programms in einem separaten Natural-Programmierobjekt, einer Local Data Area, definieren.

Ein Natural-Programmierobjekt des Typs Local Data Area (LDA) ermöglicht es Programmen, Subprogrammen, externen Subroutinen und Klassen identische Datenelement-Definitionen zu benutzen (z.B. identische Feldnamen und Formate), aber den Inhalt der Datenelemente für jedes einzelne Objekt separat zu halten.

Eine Local Data Area wird initialisiert, wenn ein Programm, Subprogramm oder eine Klasse oder externe Subroutine, das oder die diese Local Data Area benutzt, ausgeführt wird.

Im ersten Beispiel sind die Felder innerhalb des `DEFINE DATA`-Statements des Programms definiert. Im zweiten Beispiel sind dieselben Felder in einer Local Data Area definiert, und das `DEFINE DATA`-Statement enthält lediglich eine Referenz auf diese Data Area.

Beispiel 1 — Felddefinitionen innerhalb des DEFINE DATA-Statements:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

Beispiel 2 — Felddefinitionen in einer separaten Data Area:

Programm:

Das Programm selbst enthält keine Felddefinitionen, sondern referenziert die in der `LDA39` enthaltenen Felddefinitionen.

```
DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
  ...
```

Referenzierte Local Data Area `LDA39`:

```
I T L  Name                            F Length     Miscellaneous
All -- -------------------------------- - ---------- ------------------------->
  V  1 VIEWEMP                                      EMPLOYEES
     2 PERSONNEL-ID                    A          8
     2 FIRST-NAME                      A         20
     2 NAME                            A         20
     1 #VARI-A                         A         20
     1 #VARI-B                         N        3.2
     1 #VARI-C                         I          4
```

Tipp: Um eine übersichtlich strukturierte und einheitliche Anwendung zu erhalten, ist es in der Regel besser, Felder in Data Areas außerhalb der Programme zu definieren.

Global Data Area

Wenn Sie die Datenelemente in einem separaten Bereich definieren möchten, der von mehreren Natural-Programmierobjekten genutzt werden kann, verwenden Sie eine Globa Data Area.

Die folgenden Themen werden erörtert:

- GDA anlegen und referenzieren
- GDA-Instanzen
- Datenblöcke

GDA anlegen und referenzieren

GDAs werden mit dem Natural Data Area Editor angelegt und geändert. Weitere Informationen entnehmen Sie dem Abschnitt *Data Area Editor* in der *Editors*-Dokumentation.

Eine GDA, die von einem Natural-Programmierobjekt referenziert wird, muss in derselben Natural-Library (oder in einer für diese Library definierten Steplib) gespeichert werden, in der auch das diese GDA referenzierende Objekt gespeichert ist.

Wenn eine GDA mit Namen `COMMON` in einer Library vorhanden ist, wird das Programm mit Namen `ACOMMON` automatisch aufgerufen, wenn Sie sich mit einem `LOGON` in dieser Library anmelden.

Wichtig: Wenn Sie eine Anwendung erstellen, bei der mehrere Natural-Programmierobjekte eine GDA referenzieren, denken Sie bitte daran, dass Änderungen an den Datenelement-Definitionen in der GDA alle Natural-Programmierobjekte betreffen, die diese Data Area referenzieren. Deshalb müssen diese Objekte mittels des Kommandos `CATALOG` oder `STOW` neu kompiliert werden, nachdem die GDA geändert worden ist.

Um eine GDA zu benutzen, muss ein Natural-Programmierobjekt sie mit der `GLOBAL`-Klausel des `DEFINE DATA`-Statements referenzieren.

Jedes Natural-Programmierobjekt kann nur eine GDA referenzieren; d.h. ein `DEFINE DATA`-Statement darf nicht mehr als eine `GLOBAL`-Klausel enthalten.

GDA-Instanzen

Die erste Instanz einer GDA wird angelegt und zur Laufzeit initialisiert, wenn das erste, sie referenzierende Natural-Programmierobjekt ausgeführt wird.

Sobald eine GDA-Instanz angelegt worden ist, können die Datenwerte, die sie enthält, von allen Natural-Programmierobjekten gemeinsam benutzt werden, die diese GDA referenzieren (`DEFINE DATA GLOBAL`-Statement) und die von einem `PERFORM`-, `INPUT`- oder `FETCH`-Statement aufgerufen werden. Alle Objekte, die eine GDA-Instanz gemeinsam benutzen, greifen auf dieselben Datenelemente zu.

Eine neue GDA-Instanz wird erstellt, wenn Folgendes gilt:

- Ein Subprogramm, das eine GDA (eine *beliebige* GDA) referenziert, wird mit einem `CALLNAT`-Statement aufgerufen.
- Ein Subprogramm, das *keine* GDA referenziert, ruft ein Programmierobjekt auf, das eine GDA (eine *beliebige* GDA) referenziert.

Beim Anlegen einer neuen Instanz einer GDA wird die aktuelle GDA-Instanz zeitweilig unterbrochen und die Datenwerte, die sie enthält, werden in den Stack geschrieben. Das Subprogramm referenziert dann die Datenwerte in der neu erstellten GDA-Instanz. Auf die Datenwerte in der/den zeitweilig unterbrochenen GDA-Instanz/Instanzen ist kein Zugriff möglich.

Ein Programmierobjekt bezieht sich nur auf eine GDA-Instanz und kann nicht auf vorherige GDA-Instanzen zugreifen. Ein GDA-Datenelement kann nur an ein Subprogramm übergeben werden, wenn das Element als ein Parameter im `CALLNAT`-Statement definiert wird.

Wenn das Subprogramm zum aufrufenden Programmierobjekt zurückkehrt, wird die es referenzierende GDA-Instanz gelöscht, und die vorher zeitweilig unterbrochene GDA-Instanz wird mit ihren Datenwerten wieder aufgenommen.

Eine GDA-Instanz und ihr Inhalt wird gelöscht, wenn einer der folgenden Punkte gilt:

- Das nächste `LOGON` wird ausgeführt.
- Eine andere GDA wird auf derselben Stufe referenziert (Stufen sind später in diesem Abschnitt beschrieben).
- Ein `RELEASE VARIABLES`-Statement wird ausgeführt.

  In diesem Fall werden die Datenwerte in einer GDA-Instanz zurückgesetzt, und zwar entweder wenn ein Programm auf der Stufe 1 seine Ausführung beendet, oder wenn das Programm ein anderes Programm über ein `FETCH`- oder `RUN`-Statement aufruft.

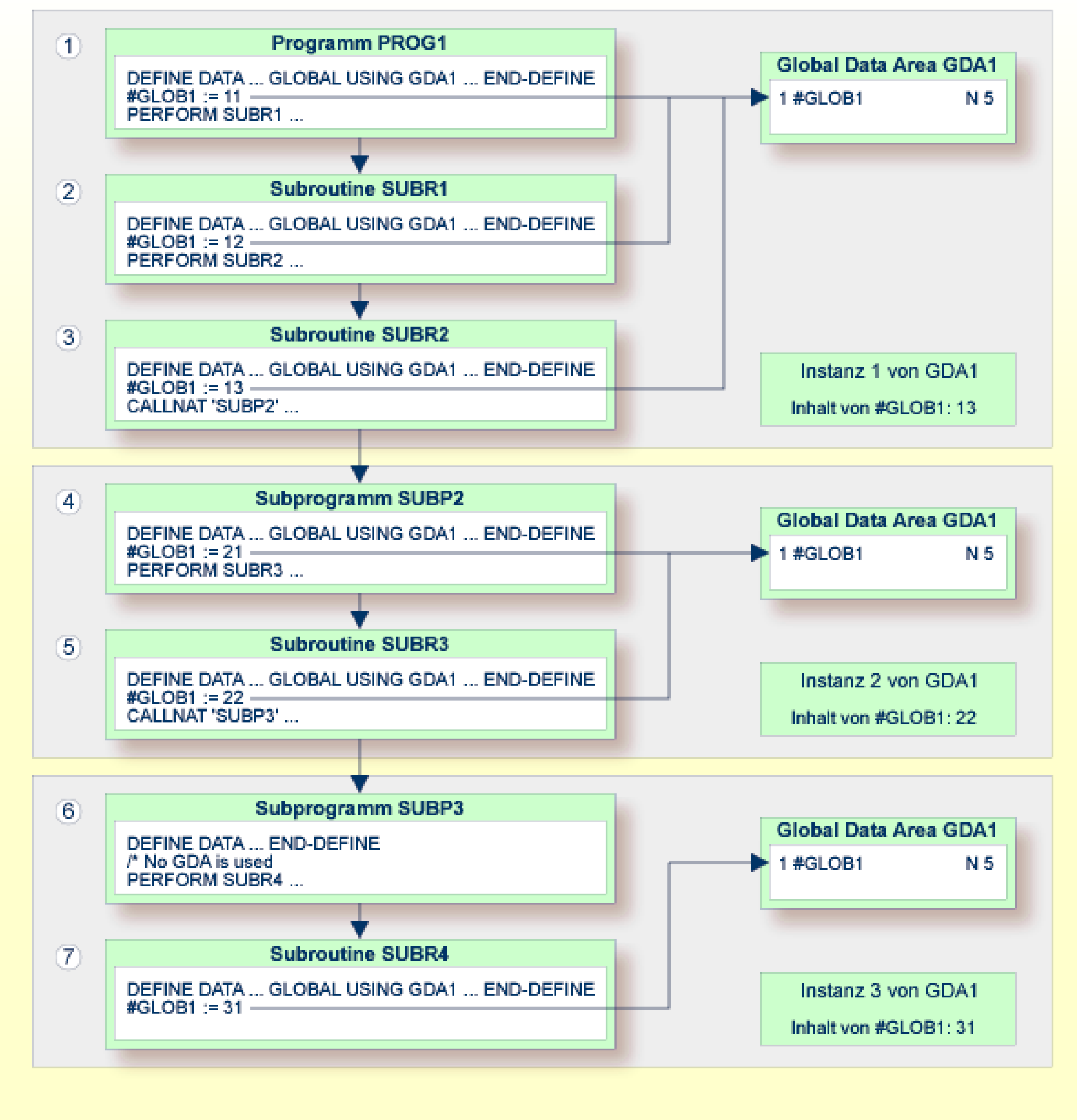
  Die folgende Grafik zeigt, wie Programmierobjekte GDAs referenzieren und Datenelemente in GDA-Instanzen mitbenutzen.

GDA-Instanzen gemeinsam benutzen

Die Grafik weiter unten veranschaulicht, dass ein eine GDA referenzierendes Subprogramm die Datenwerte in einer GDA-Instanz nicht gemeinsam benutzen kann, die von dem aufrufenden Programm referenziert wird.

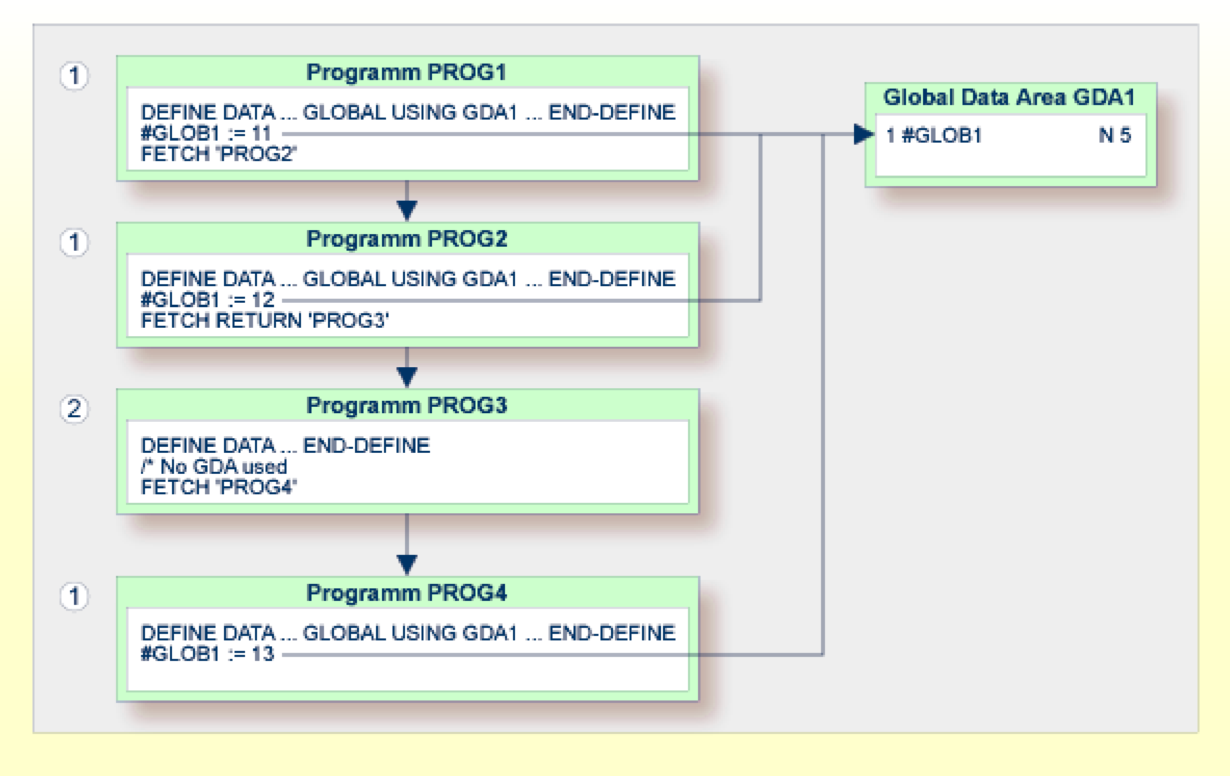
Ein Subprogram, das dieselbe GDA referenziert wie das aufrufende Programm, erstellt eine neue Instanz dieser GDA. Die Datenelemente, die in einer GDA definiert sind, die von einem Subprogramm referenziert wird, können aber von einer vom Subprogramm aufgerufenen Subroutine oder Helproutine gemeinsam benutzt werden.

Die folgende Grafik zeigt drei GDA-Instanzen von GDA1 und die Endwerte, die jeder GDA-Instanz vom Datenelement #GLOB1 zugewiesen werden. Die Zahlen 1 bis 7 verweisen auf die hierarchischen Stufen der Programmierobjekte.



Die folgende Grafik veranschaulicht, dass Programme, die dieselbe GDA referenzieren und sich gegenseitig mit dem `FETCH`- oder `FETCH RETURN`-Statement aufrufen, die in dieser GDA definierten Datenelemente gemeinsam benutzen. Wenn eines dieser Programme keine GDA referenziert, bleibt die Instanz der vorher referenzierten GDA aktiv, und die Werte der Datenelemente werden zurückbehalten.

Die Ziffern 1 und 2 verweisen auf die hierarchischen Stufen der Programmierobjekte.



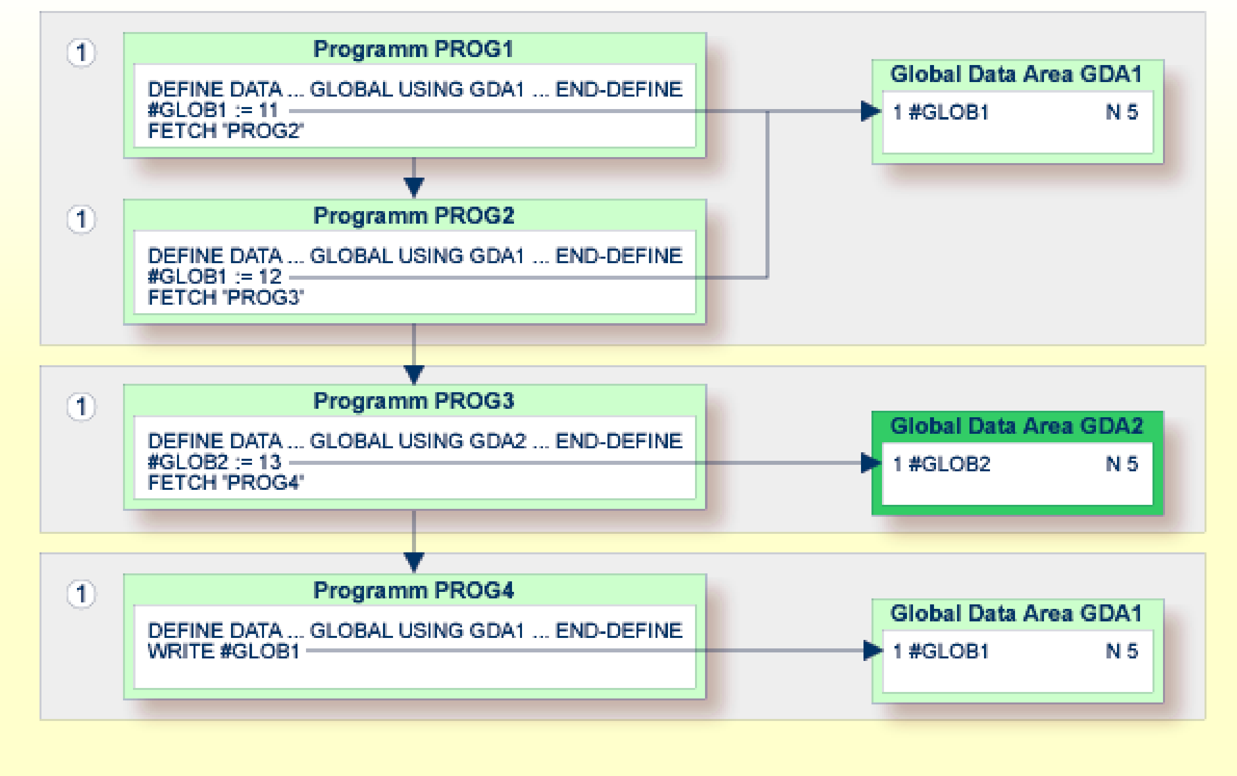
Die folgende Grafik veranschaulicht, dass, wenn ein Programm das `FETCH`-Statement benutzt, um ein anderes Programm aufzurufen, das eine unterschiedliche GDA referenziert, wird die aktuelle, vom aufrufenden Programm referenzierte Instanz der GDA (hier: GDA1) gelöscht. Wenn diese GDA dann erneut von einem anderen Programm referenziert wird, eine neue Instanz dieser GDA erstellt wird, bei der alle Datenelemente ihre Anfangswerte haben.

Sie können das `FETCH RETURN`-Statement nicht benutzen, um ein anderes Programm aufzurufen, das eine unterschiedliche GDA referenziert.

Die Ziffer 1 verweist auf die hierarchische Stufe der Programmierobjekte.

Die aufrufenden Programme `PROG3` und `PROG4` beeinflussen die GDA-Instanzen wie folgt:

- Das Statement `GLOBAL USING GDA2` in `PROG3` erstellt eine Instanz von `GDA2` und löscht die aktuelle Instanz von `GDA1`.
- Das Statement `GLOBAL USING GDA1` in PROG4 löscht die aktuelle Instanz von `GDA2` und erstellt eine neue Instanz von `GDA1`. Als Ergebnis davon zeigt das `WRITE`-Statement den Wert Null (0) an.



Datenblöcke

Um Datenspeicher zu sparen, können Sie eine GDA mit Datenblöcken erstellen.

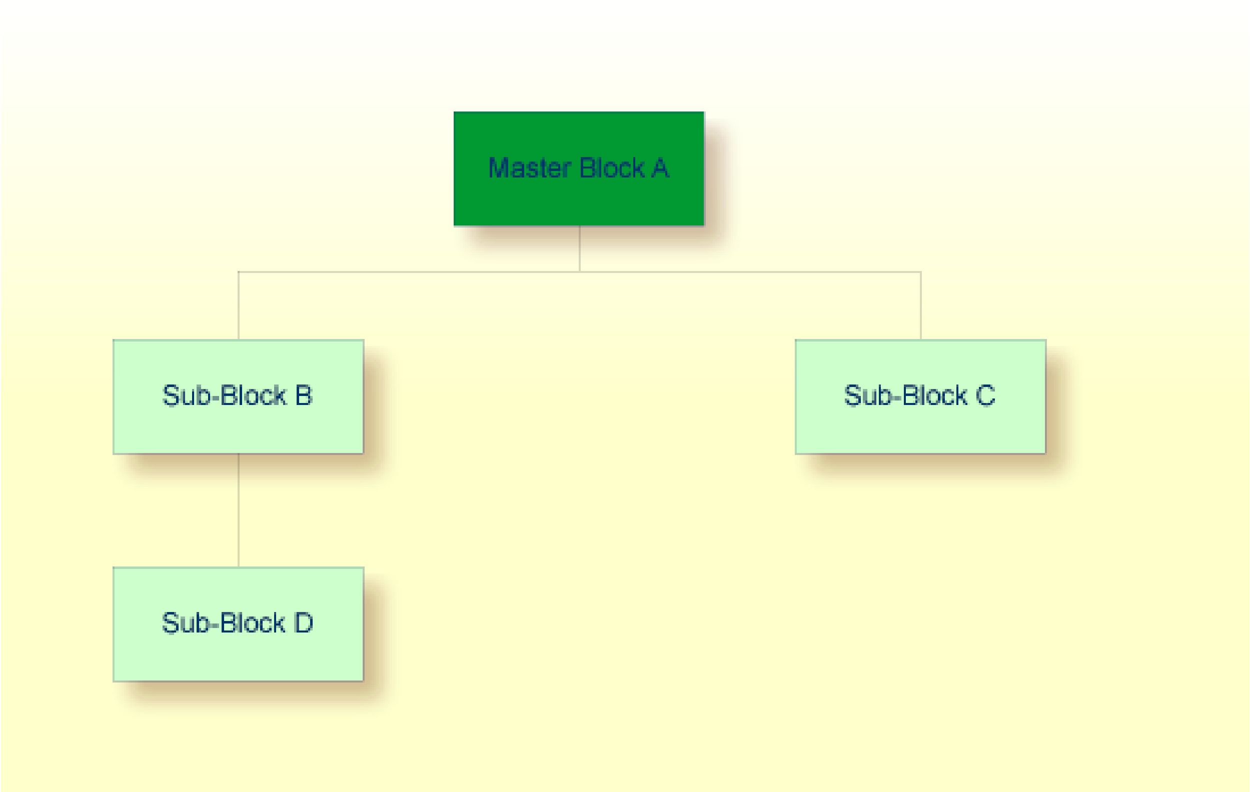
Folgende Themen werden in diesem Abschnitt behandelt:

- Beispiel für die Benutzung von Datenblöcken
- Datenblöcke definieren
- Block-Hierarchien

Beispiel für die Benutzung von Datenblöcken

Datenblöcke können sich bei der Programmausführung gegenseitig überlagern, was zu einem Einsparen von Speicherplatz führt.

Gehen wir beispielsweise davon aus, dass bei der folgenden vorgegebenen Hierarchie den Blöcken B und C derselbe Speicherbereich zugewiesen würde. So wäre es nicht möglich, dass die Blöcke B und C gleichzeitig in Benutzung sind. Die Änderung von Block B würde zur Zerstörung des Inhalts von Block C führen.



Datenblöcke definieren

Datenblöcke werden im Data-Area-Editor definiert. Sie bauen die Block-Hierarchie auf, indem Sie angeben, welcher Block welchem anderen untergeordnet ist: geben Sie dazu den Namen des übergeordneten „Parent"-Blocks in das Kommentarfeld der Block-Definition ein.

In dem folgenden Beispiel sind `SUB-BLOCKB` und `SUB–BLOCKC` dem Block `MASTER–BLOCKA` untergeordnet; `SUB–BLOCKD` ist `SUB–BLOCKB` untergeordnet.

Die maximale Anzahl der Block-Stufen ist 8 (einschießlich des Master-Blocks).

Beispiel:

Global Data Area `G-BLOCK`:

```
I T L Name                             F Leng Index/Init/EM/Name/Comment
- - - -------------------------------- - ---- ---------------------------------
  B   MASTER-BLOCKA
    1 MB-DATA01                        A   10
  B   SUB-BLOCKB                              MASTER-BLOCKA
    1 SBB-DATA01                       A   20
  B   SUB-BLOCKC                              MASTER-BLOCKA
    1 SBC-DATA01                       A   40
  B   SUB-BLOCKD                              SUB-BLOCKB
```

```
      1 SBD-DATA01                       A    40
```

Um die spezifischen Blöcke einem Programm zur Verfügung zu stellen, benutzen Sie die folgende Syntax im `DEFINE DATA`-Statement:

Programm 1:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA
END-DEFINE
```

Programm 2:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
```

Programm 3:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKC
END-DEFINE
```

Programm 4:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB.SUB-BLOCKD
END-DEFINE
```

Mit dieser Struktur kann Programm 1 die Daten in `MASTER–BLOCKA` gemeinsam mit Programm 2, Programm 3 oder Programm 4 benutzen. Allerdings können die Programme 2 und 3 nicht dieselben Data Areas von `SUB–BLOCKB` und `SUB–BLOCKC` gemeinsam benutzen, weil diese Datenblöcke auf derselben Struktur-Stufe definiert sind, und folglich denselben Speicherbereich belegen.

Block-Hierarchien

Besonders sorgfältig müssen Sie vorgehen, wenn Sie Datenblock-Hierarchien benutzen. Gehen wir von folgendem Szenario mit drei Programmen aus, die eine Datenblock-Hierarchie verwenden:

Programm 1:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
MOVE 1234 TO SBB-DATA01
FETCH 'PROGRAM2'
END
```

Programm 2:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA
END-DEFINE
*
FETCH 'PROGRAM3'
END
```

Programm 3:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
WRITE SBB-DATA01
END
```

Erläuterung:

- Programm 1 benutzt die Global Data Area `G-BLOCK` mit `MASTER-BLOCKA` und `SUB-BLOCKB`. Das Programm ändert ein Feld in `SUB-BLOCKB` und ruft Programm 2 mit einem `FETCH`-Statement auf, welches nur `MASTER-BLOCKA` in seiner Datendefinition angegeben hat.
- Programm 2 setzt `SUB-BLOCKB` zurück (löscht den Inhalt von `SUB-BLOCKB`). Der Grund dafür ist, dass ein Programm auf Stufe 1 (zum Beispiel ein mit einem `FETCH`-Statement aufgerufenes Programm) alle Datenblöcke zurücksetzt, die den Blöcken untergeordnet sind, die es in seiner eigenen Datendefinition festlegt.

- Programm 2 ruft jetzt Programm 3 mit einem `FETCH`-Kommando auf, welches das in Programm 1 geänderte Feld anzeigen soll, aber es gibt einen leeren Bildschirm zurück.

Einzelheiten zu den Programmfstufen entnehmen Sie dem Abschnitt ***Mehrere Stufen (Levels) aufgerufener Objekte***.

Parameter Data Area

Ein Natural-Programmierobjekt des Typs Parameter Data Area (PDA) wird benutzt, um die Datenelemente zu definieren, die als Parameter an ein Subprogramm, eine externe Subroutine oder Helproutine übergeben werden.

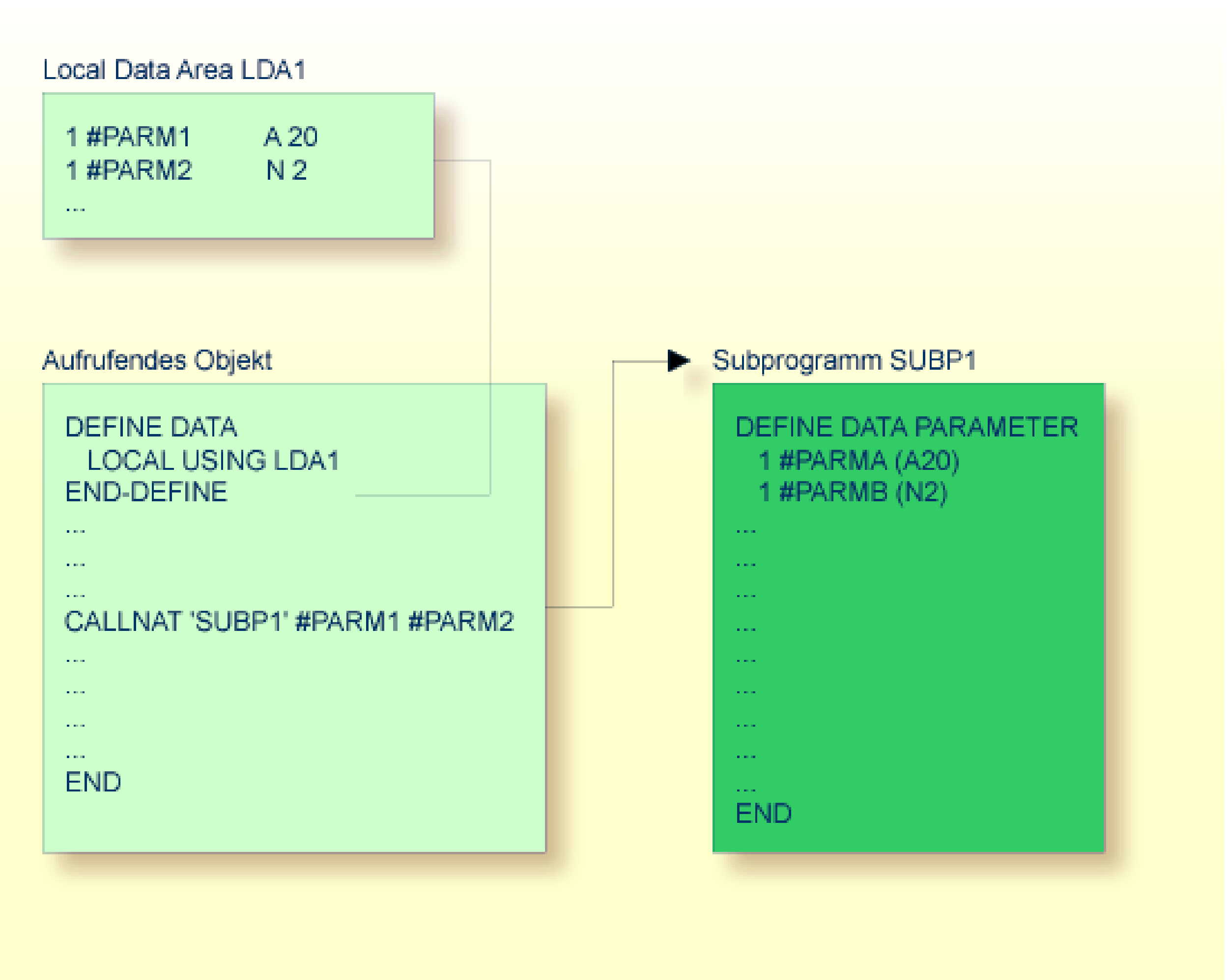
Eine PDA ermöglicht es Subprogrammen, externen Subroutinen und Helproutinen, dieselben Datenelement-Definitionen zu benutzen (zum Beispiel, identische Feldnamen und -formate).

Ein Subprogramm wird mit einem `CALLNAT`-Statement aufgerufen. Mit dem `CALLNAT`-Statement können Parameter von dem aufrufenden Objekt an das Subprogramm übergeben werden.

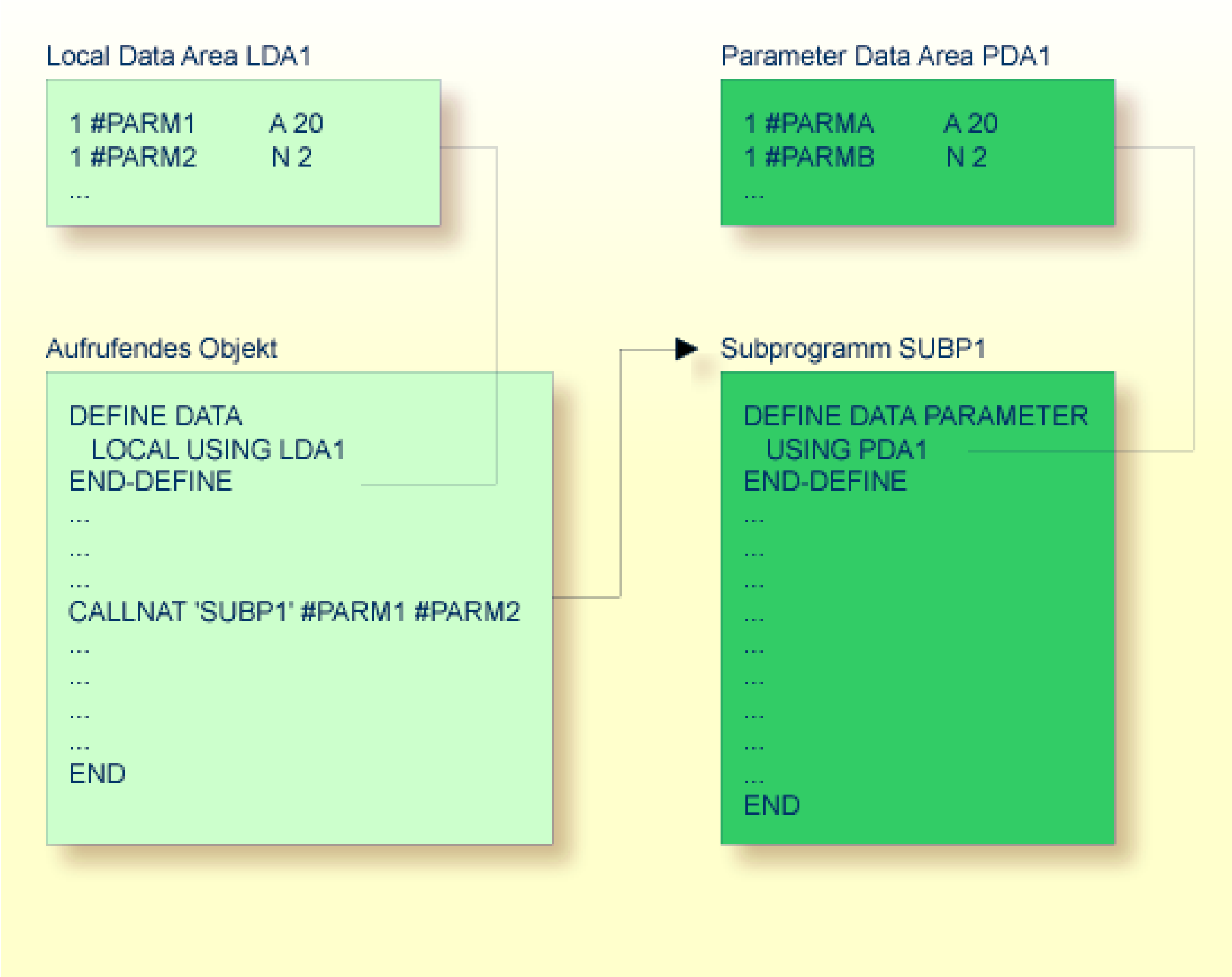
Diese Parameter müssen im Subprogramm in einem `DEFINE DATA PARAMETER`-Statement definiert werden:

- sie können entweder in der `PARAMETER`-Klausel des `DEFINE DATA`-Statements selbst definiert werden oder
- sie können definiert werden in einer separaten Parameter Data Area, die von dem `DEFINE DATA PARAMETER`-Statement referenziert wird.

Innerhalb des DEFINE DATA PARAMETER-Statements definierte Parameter



Separat in einer Parameter Data Area definierte Parameter



In der gleichen Weise wie beim `CALLNAT`-Statement müssen Parameter, die mit einem `PERFORM`-Statement an eine externe Subroutine übergeben werden, in der externen Subroutine in einem `DEFINE DATA PARAMETER`-Statement definiert werden.

Im aufrufenden Objekt müssen die an das Subprogramm bzw. die Subroutine übergebenen Parametervariablen nicht in einer Parameter Data Area definiert werden; in der obigen Abbildung sind sie in einer vom aufrufenden Objekt benutzten Local Data Area definiert (man hätte sie aber auch in einer Global Data Area definieren können).

Reihenfolge, Format und Länge der im `CALLNAT`- bzw. `PERFORM`-Statement des aufrufenden Objekts angegebenen Parameter müssen genau mit Reihenfolge, Format und Länge der Felder, die im `DEFINE DATA PARAMETER`-Statement des aufgerufenen Subprogramms bzw. der aufgerufenen Subroutine definiert sind, übereinstimmen.

Die Namen der Variablen im aufrufenden Objekt und dem aufgerufenen Subprogramm bzw. der aufgerufenen Subroutine brauchen nicht dieselben zu sein (da die Übergabe der Parameter nach Speicheradressen erfolgt und nicht nach Namen).

Um zu garantieren, dass die im aufrufenden Programm benutzten Datenelement-Definitionen mit den im Subprogramm oder der externen Subroutine benutzten Datenelement-Definitionen identisch sind, können Sie eine PDA in einem `DEFINE DATA LOCAL USING`-Statement angeben. Wenn Sie eine PDA als eine LDA benutzen, können Sie sich den zusätzlichen Aufwand der Erstellung einer LDA ersparen, die dieselbe Struktur wie die PDA hat.

6 Programme, Functions, Subprogramme und Subroutinen

- Modulare Anwendungsstruktur ... 36
- Mehrere Stufen (Levels) aufgerufener Objekte ... 36
- Programm ... 38
- Function ... 41
- Subroutine ... 43
- Subprogramm ... 48
- Verarbeitungsablauf beim Aufruf eines Unterprogramms ... 50

Dieses Dokument liefert Informationen über die Objekttypen, die als Routinen, d.h. als untergeordnete Programme aufgerufen werden können.

Anmerkung: Obwohl sie auch von anderen Objekten aufgerufen werden, sind Helproutinen und Maps genau genommen keine Routinen als solche, und werden deshalb in getrennten Dokumenten beschrieben; siehe Abschnitt ***Helproutinen***.

Modulare Anwendungsstruktur

Eine typische Natural-Anwendung besteht nicht aus einem einzigen großen Programm, sondern ist in mehrere Module aufgeteilt. Jedes dieser Module stellt eine funktionale Einheit von überschaubarer Größe dar, und jedes Modul ist mit den anderen Modulen der Anwendung auf eine klar definierte Weise verbunden. Dadurch ergibt sich eine übersichtlich strukturierte Anwendung, was die Entwicklung und anschließende Wartung erheblich erleichtert und beschleunigt.

Ein Hauptprogramm, das ausgeführt wird, kann andere Programme, Subprogramme, Subroutinen, Helproutinen und Maps aufrufen. Diese Objekte können ihrerseits wiederum andere Objekte aufrufen (eine Subroutine kann beispielsweise eine andere Subroutine aufrufen). Dadurch kann die modulare Struktur einer Anwendung äußerst komplex und vielschichtig werden.

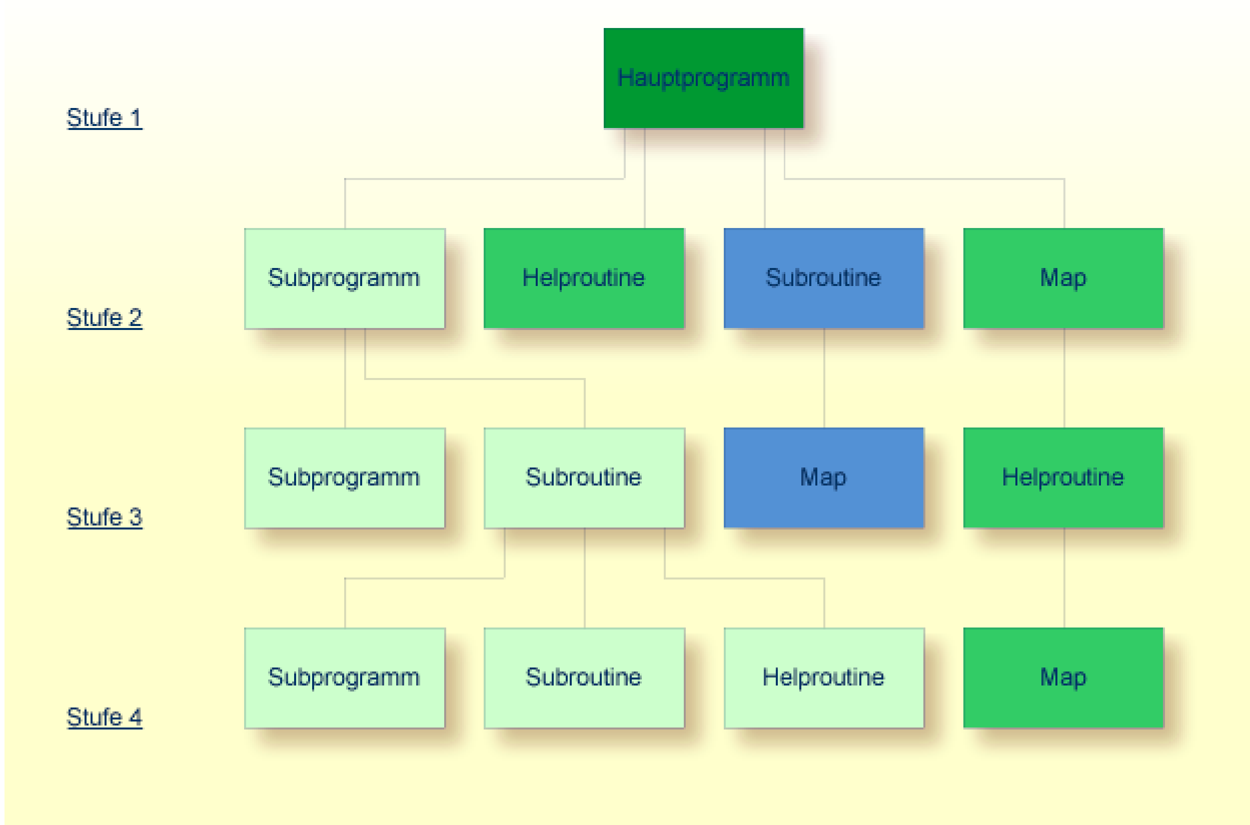
Mehrere Stufen (Levels) aufgerufener Objekte

Ein aufgerufenes Objekt ist jeweils eine Stufe (Level) unter dem Objekt, von dem es aufgerufen wurde; d.h. mit jedem Aufruf eines untergeordneten Objekts erhöht sich die Stufennummer um 1.

Ein Programm, das selbständig ausgeführt wird, wird auf Stufe 1 eingeordnet; Subprogramme, Subroutinen, Maps oder Helproutinen, die direkt von diesem Hauptprogramm aufgerufen werden, sind auf Stufe 2; ruft eine dieser Subroutinen ihrerseits eine andere Subroutine auf, so ist letztere auf Stufe 3.

Wird von einem Objekt über ein `FETCH`-Statement ein anderes Programm aufgerufen, so wird dies als Hauptprogramm eingestuft und auf Stufe 1 eingeordnet. Ein Programm, das mit `FETCH RETURN` aufgerufen wird, wird dagegen als Unterprogramm eingestuft und ist eine Stufe unter dem Objekt, von dem es aufgerufen wurde.

Die folgende Abbildung enthält ein Beispiel für mehrere Stufen (Levels) aufgerufener Objekte und zeigt, wie diese Stufen gezählt werden:



Um die Level-Nummer des Objekts, das gerade ausgeführt wird, zu erfahren, können Sie die Systemvariable `*LEVEL` verwenden (die in der *Systemvariablen*-Dokumentation beschrieben ist).

Der vorliegende Abschnitt behandelt folgende Natural-Objekttypen, die als Unterprogramme (d.h. als untergeordnete Programme) aufgerufen werden können:

- Programm
- Function
- Subroutine
- Subprogramm

Helproutinen und Maps werden zwar auch von anderen Objekten aufgerufen, sind aber keine Unterprogramme im eigentlichen Sinne und werden daher in separaten Abschnitten behandelt, siehe ***Helproutinen*** und ***Maps***.

Grundsätzlich unterscheiden sich Programme, Subprogramme und Subroutinen dadurch voneinander, wie Daten zwischen ihnen übergeben werden können und welche Data Areas sie gemeinsam

benutzen können. Die Entscheidung, welchen Objekttyp Sie verwenden, ergibt sich daher im wesentlichen aus der Datenstruktur Ihrer Anwendung.

Programm

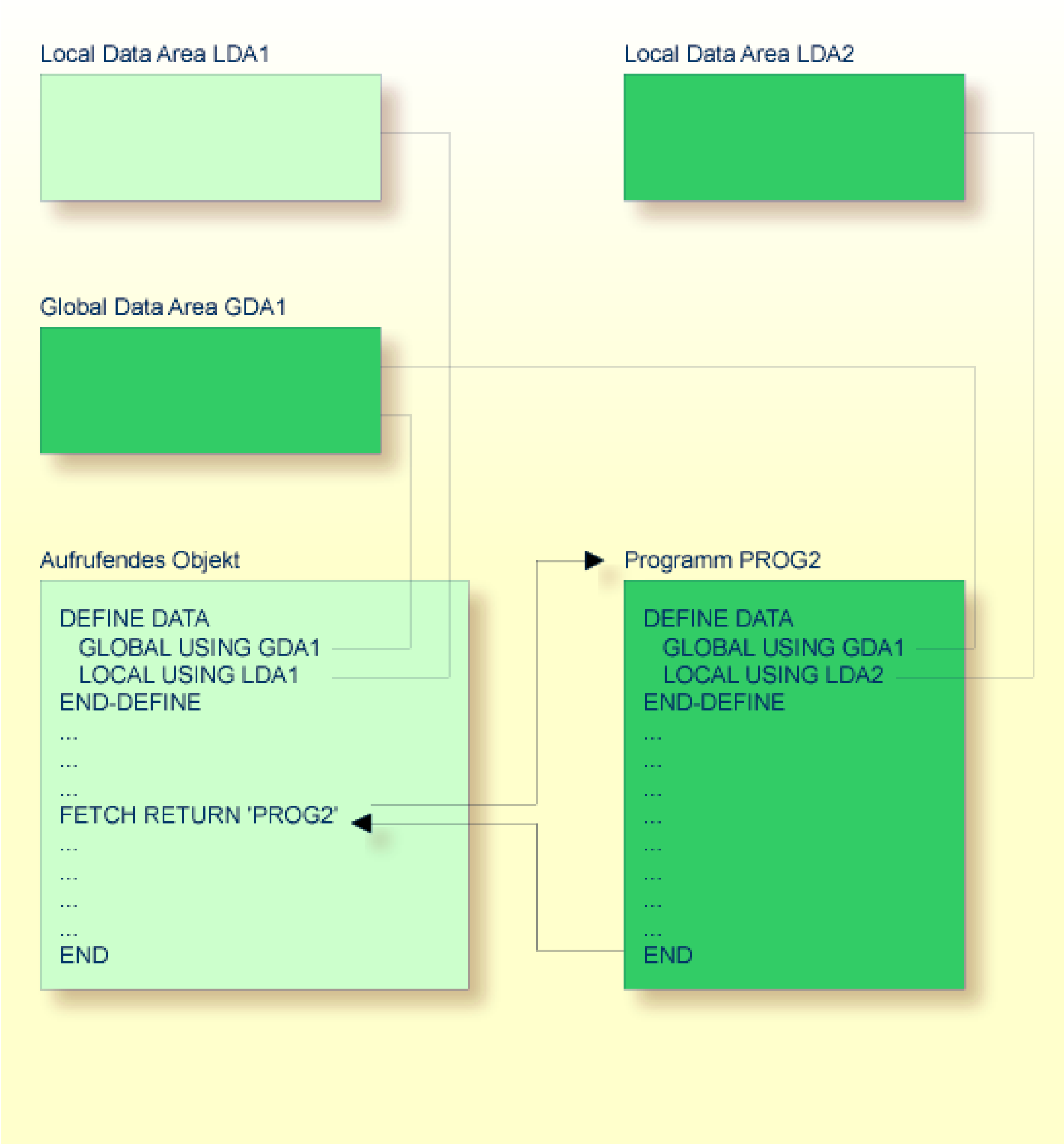
Ein Programm kann selbständig ausgeführt — und getestet — werden.

- Um ein Source-Programm zu kompilieren und anschließend auszuführen, verwenden Sie das Systemkommando `RUN`.
- Um ein Programm auszuführen, das bereits in kompilierter Form existiert, verwenden Sie das Systemkommando `EXECUTE`.

Ein Programm kann von einem anderen Objekt mit einem `FETCH`- oder `FETCH RETURN`-Statement aufgerufen werden. Das aufrufende Objekt kann ein Programm, ein **Subprogramm**, **eine Function**, eine **Subroutine** oder eine **Helproutine** sein.

- Wenn ein Programm mit `FETCH RETURN` aufgerufen wird, wird die Ausführung des aufrufenden Objekts unterbrochen — nicht beendet —, und das aufgerufene Programm wird als *Unterprogramm* aktiviert. Wenn die Ausführung des aufgerufenen Programms beendet ist, wird das aufrufende Objekt reaktiviert und seine Ausführung mit dem nächsten Statement nach dem `FETCH RETURN`-Statement fortgesetzt.
- Wenn ein Programm mit `FETCH` aufgerufen wird, wird die Ausführung des aufrufenden Objekts beendet, und das aufgerufene Programm wird als *Hauptprogramm* aktiviert. Das aufrufende Objekt wird nach beendeter Ausführung des aufgerufenen Programms nicht reaktiviert.

Mit FETCH RETURN aufgerufenes Programm

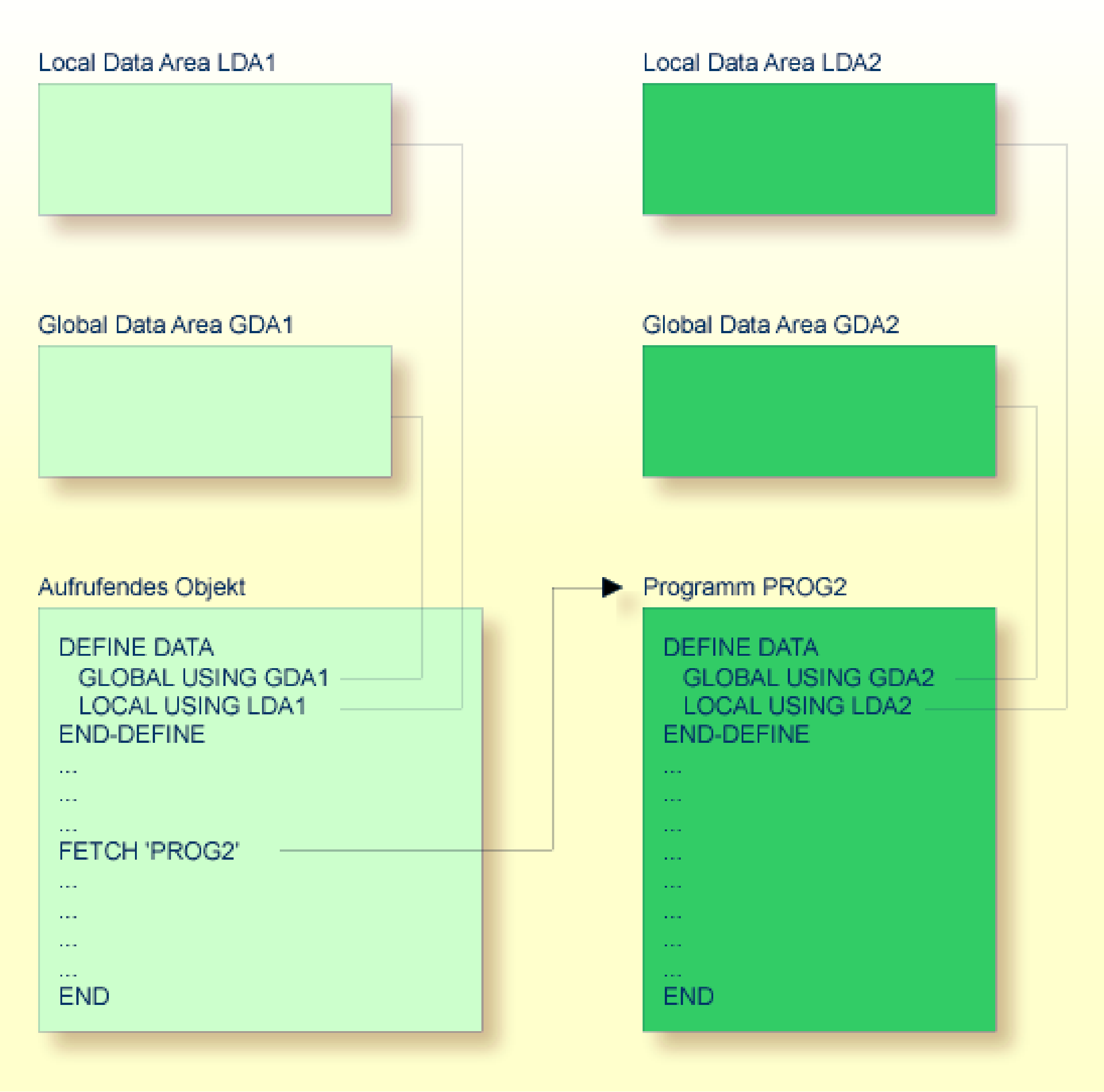


Ein mit `FETCH RETURN` aufgerufenes Programm kann auf die vom aufrufenden Objekt benutzte Global Data Area zugreifen.

Darüber hinaus kann jedes Programm seine eigene Local Data Area haben, in der die nur in diesem Programm verwendeten Felder definiert sind.

Ein mit `FETCH RETURN` aufgerufenes Programm kann jedoch keine eigene Global Data Area haben.

Mit FETCH aufgerufenes Programm



Ein mit `FETCH` als Hauptprogramm aufgerufenes Programm verwendet in der Regel seine eigene Global Data Area (wie in der obigen Abbildung gezeigt). Es könnte allerdings auch dieselbe Global Data Area verwenden wie das aufrufende Objekt.

Anmerkung: Ein Source-Programm kann auch mit einem `RUN`-Statement aufgerufen werden; siehe `RUN`-Statement im der *Statements*-Dokumentation.

Function

Dieses Dokument beschreibt die Vorteile der Verwendung des Programmierobjekts „Function", erläutert die Unterschiede bei der Verwendung von „Function Calls" und „Subprogram Calls" und stellt die verfügbaren Methoden zum Definieren und Aufrufen einer „Function" vor.

Ein Objekt des Typs „Function" enthält die Definitionen einer einzelnen Funktion und kann wie im folgenden Code-Beispiel gezeigt aufgebaut sein:

```
DEFINE FUNCTION
   ...
   DEFINE SUBROUTINE
   ...
   END-SUBROUTINE
   ...
END-FUNCTION
```

Der Statements-Block zwischen `DEFINE FUNCTION` und `END-FUNCTION` muss alle Statements enthalten, die ausgeführt werden sollen, wenn die Function aufgrufen wird.

Innerhalb der Function-Definition dürfen auch interne Subroutinen defineirt werden.

Zum Aufrufen einer Function wird die **Function Call**-Syntax verwendet.

Wenn Sie eine Code-Block haben, der innerhlab des Objekts mehrmals ausgeführt werden soll, ist es von Nutzen, eine interne Subroutine zu verwenden. Dann brauchen Sie diesen Block nur einmal in einem `DEFINE SUBROUTINE`-Statement-Bock zu codieren und können ihn mit mehreren `PERFORM`-Statements aufrufen.

Die **Global Data Area** des aufrufenden Objekts (z.B. `GDA1`) kann nicht in der Function-Definition referenziert werden. Darüber hinaus können Objekte, die von einer Function aufgerufen werden, nicht die Global Data Area (GDA) des die Function aufrufenden Objekts (`GAD1`) aufrufen, weil die Laufzeitumgebung zu Beginn einer Function eine neue Global Data Area erstellt.

Parameter Data Areas (z.B. `PDA1`) können Sie benutzen, um auf Parameter für Function Calls und auf Function-Definitionen zuzugreifen, um so den Pflegeaufwand zu minimieren, falls die Parameter geändert werden müssen.

Das **Copycode**-Objekt, das die Prototype-Definition enthält, wird nur beim Kompilieren benutzt, um den Typ der Return-Variablen für die Function-Call-Referenz zu bestimmen und um, falls gewünscht, die Parameter zu prüfen.

Subroutine

Die Statements, aus denen eine Subroutine besteht, müssen innerhalb eines `DEFINE SUBROUTINE` ... `END-SUBROUTINE`-Statement-Blocks definiert werden.

Eine Subroutine wird mit einem `PERFORM`-Statement aufgerufen.

Eine Subroutine kann eine *interne Subroutine* oder eine *externe Subroutine* sein:

- **Interne Subroutine**
  Eine interne Subroutine wird innerhalb des Objekts, das das sie aufrufende `PERFORM`-Statement enthält, definiert.
- **Externe Subroutine**
  Eine externe Subroutine wird als separates Objekt - vom Typ Subroutine - außerhalb des Objektes, das sie aufruft, definiert.

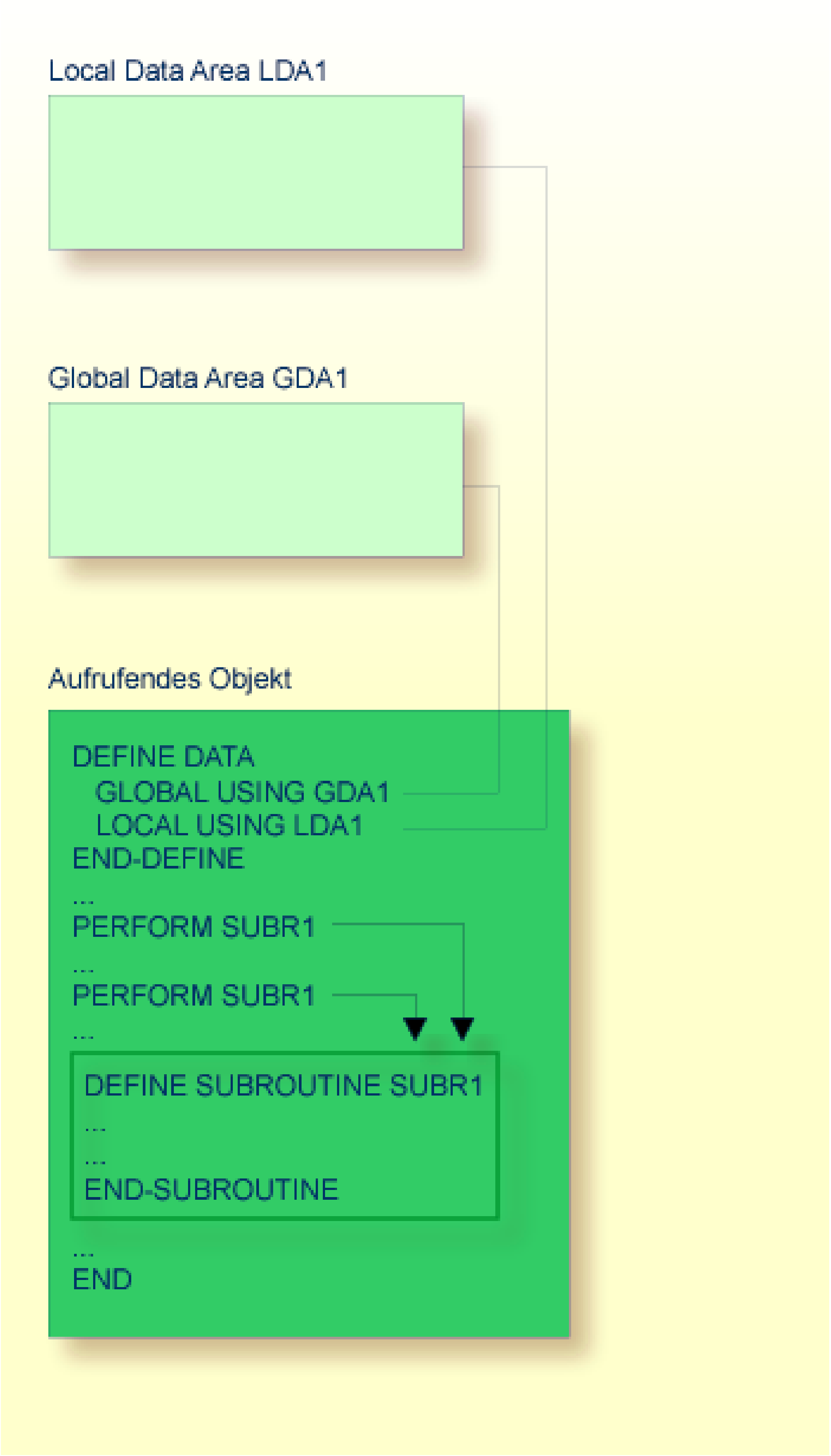
Falls Sie einen Code-Block haben, der innerhalb eines Objekts mehrmals ausgeführt werden soll, ist es sinnvoll, eine interne Subroutine zu verwenden. Sie müssen diesen Block dann nur einmal innerhalb eines `DEFINE SUBROUTINE`-Statement-Blocks kodieren, und rufen ihn dann mit mehreren `PERFORM`-Statements auf.

Die folgenden Themen werden nachfolgend erörtert:

- Interne Subroutine
- Welche Daten einer internen Subroutine zur Verfügung stehen
- Externe Subroutine

- Welche Daten einer externen Subroutine zur Verfügung stehen

Interne Subroutine


Local Data Area LDA1
Global Data Area GDA1
Aufrufendes Objekt
DEFINE DATA
GLOBAL USING GDA1
LOCAL USING LDA1
END-DEFINE
...
PERFORM SUBR1
...
PERFORM SUBR1
...
DEFINE SUBROUTINE SUBR1
...
...
END-SUBROUTINE
...
END

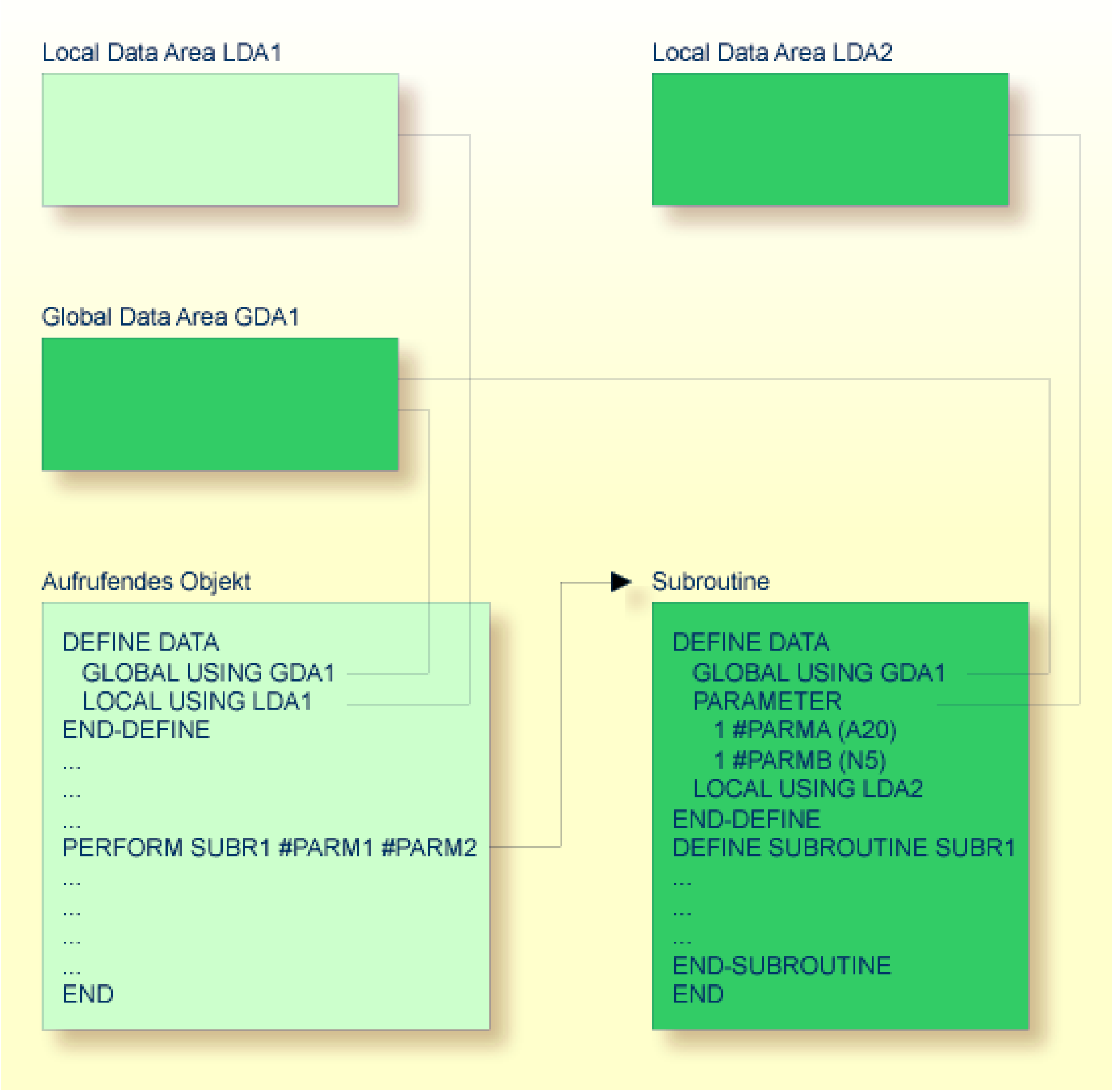
Eine interne Subroutine kann in einem Programmierobjekt vom Typ Programm, **eine Function**, Subprogramm, Subroutine oder Helproutine enthalten sein.

Wenn eine interne Subroutine so groß ist, dass sie die Lesbarkeit des Objekts, in dem sie enthalten ist, beeinträchtigt, kann es ratsam sein, sie in einer externen Subroutine unterzubringen, um die Lesbarkeit der Anwendung zu verbessern.

Welche Daten einer internen Subroutine zur Verfügung stehen

Eine interne Subroutine kann auf die **Local Data Area** und die **Global Data Area** des Objektes, in dem sie enthalten ist, zugreifen.

Externe Subroutine



Eine externe Subroutine — also ein *Objekt* vom Typ Subroutine — kann nicht selbständig ausgeführt werden. Sie muss von einem anderen Objekt aufgerufen werden. Das aufrufende Objekt kann ein Programm, **eine Function**, ein Subprogramm, eine Subroutine oder eine Helproutine sein.

Welche Daten einer externen Subroutine zur Verfügung stehen

Eine externe Subroutine kann auf die **Global Data Area** des aufrufenden Objekts zugreifen.

Darüber hinaus können mit dem `PERFORM`-Statement Parameter von dem aufrufenden Objekt an die externe Subroutine übergeben werden. Diese Parameter müssen entweder im `DEFINE DATA PARAMETER`-Statement der Subroutine oder in einer von der Subroutine genutzen **Parameter Data Area** definiert werden.

Außerdem kann eine externe Subroutine eine eigene **Local Data Area** haben, in der die Felder definiert sind, die nur innerhalb der Subroutine verwendet werden.

Eine externe Subroutine kann jedoch keine eigene Global Data Area haben.

Subprogramm

Ein Subprogramm enthält in der Regel eine allgemein verfügbare Standardfunktion, die von verschiedenen Objekten in einer Anwendung benutzt wird.

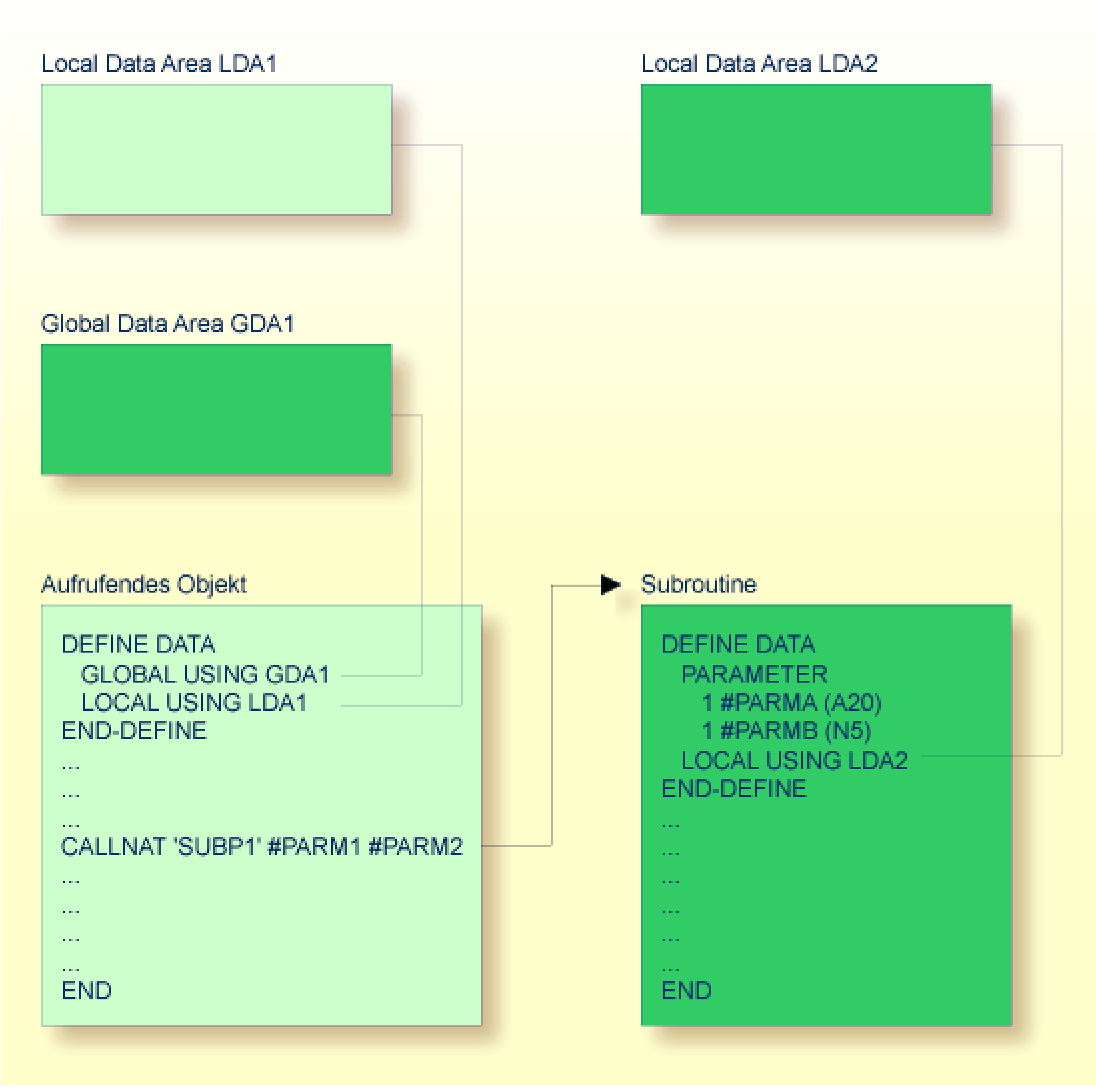
Ein Subprogramm kann nicht selbständig ausgeführt werden. Es muss von einem anderen Objekt aufgerufen werden. Das aufrufende Objekt kann ein Programm, **eine Function**, ein Subprogramm, eine Subroutine oder eine Helproutine sein.

Ein Subprogramm wird mit einem `CALLNAT`-Statement aufgerufen.

Wenn das `CALLNAT`-Statement ausgeführt wird, wird die Ausführung des aufrufenden Objekts unterbrochen und das Subprogramm ausgeführt. Nach der Ausführung des Subprogramms wird die Ausführung des aufrufenden Objekts mit dem nächsten Statement nach dem `CALLNAT`-Statement fortgesetzt.

Welche Daten einem Subprogramm zur Verfügung stehen

Mit dem `CALLNAT`-Statement können Parameter von dem aufrufenden Objekt an das Subprogramm übergeben werden. Diese Parameter sind die einzigen Daten, die dem Subprogramm vom aufrufenden Objekt zur Verfügung stehen. Sie müssen entweder im `DEFINE DATA PARAMETER`-Statement des Subprogramms oder in einer vom Subprogramm genutzen **Parameter Data Area** definiert werden.



Außerdem kann ein Subprogramm eine eigene Local Data Area haben, in der die Felder definiert sind, die innerhalb des Subprogramms verwendet werden.

Wenn ein Subprogramm seinerseits eine Subroutine oder Helproutine aufruft, kann es eine eigene **Global Data Area** haben und diese gemeinsam mit der Subroutine/Helproutine nutzen.

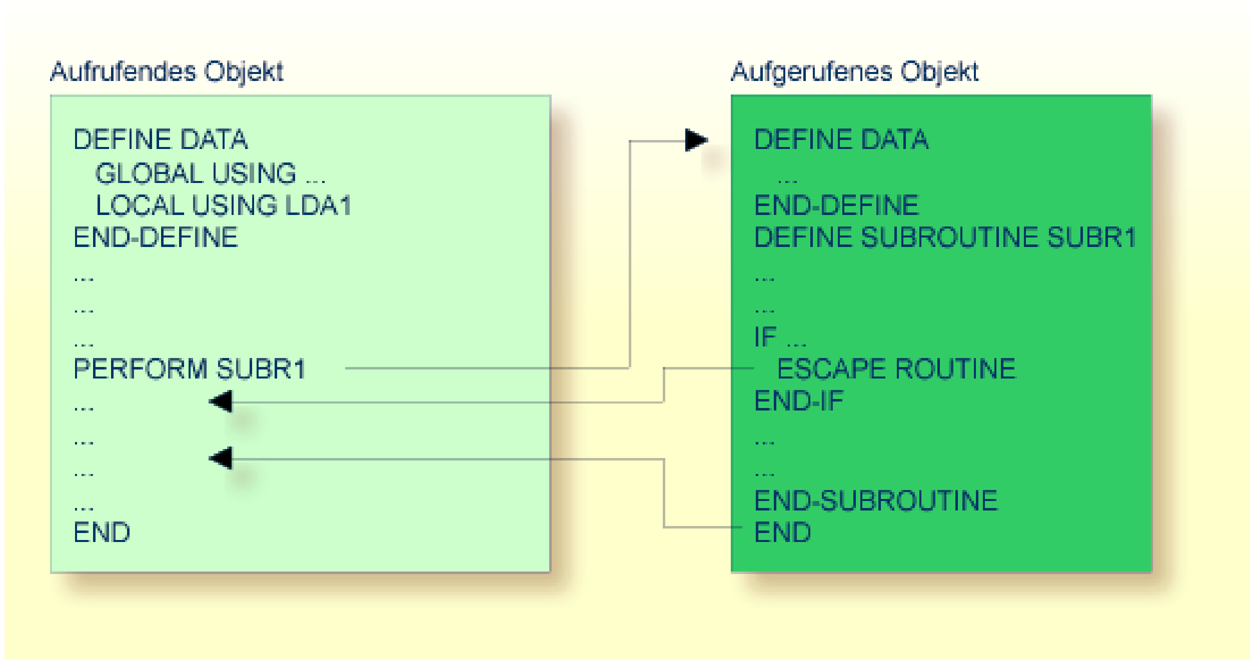
Verarbeitungsablauf beim Aufruf eines Unterprogramms

Wenn ein CALLNAT-, PERFORM- oder FETCH RETURN-Statement, das ein Unterprogramm — ein Subprogram, eine externe Subroutine bzw. ein Programm — aufruft, ausgeführt wird, wird die Ausführung des aufrufenden Objekts unterbrochen, und die Ausführung des Unterprogramms beginnt.

Die Ausführung des Unterprogramms wird fortgesetzt, bis entweder sein END-Statement erreicht ist oder die Verarbeitung des Unterprogramms durch die Ausführung eines ESCAPE ROUTINE-Statements gestoppt wird.

In beiden Fällen wird die Verarbeitung des aufrufenden Objekts mit dem nächsten Statement nach dem CALLNAT-, PERFORM- bzw. FETCH RETURN-Statement, mit dem das Unterprogramm aufgerufen wurde, fortgesetzt.

Beispiel:



7 Verarbeitung einer Rich GUI Page - Adapter

Das Natural-Objekt vom Typ Adapter können Sie verwenden, um eine Rich-GUI-Page in einer Natural-Anwendung darzustellen. Dieser Objekttyp spielt bei der Verarbeitung einer Rich-GUI-Page eine ähnliche Rolle wie der Objekttyp Map bei der Verarbeitung einer Terminal-Eingabe/Ausgabe. Im Gegensatz zur Map enthält ein Adapter jedoch keine Layout-Informationen.

Ein Objekt vom Typ Adapter wird aus einem externen Page-Layout erzeugt. Es dient als Schnittstelle, über die eine Natural-Anwendung unter Verwendung eines extern definierten und gespeicherten Page-Layouts Daten an ein externes Eingabe-/Ausgabesystem senden kann. Das Objekt Adapter enthält den zum Durchführen dieser Aufgabe erforderlichen Natural-Code.

Eine Anwendung referenziert einen Adapter in einem `PROCESS PAGE USING`-Statement.

Weitere Informationen zum Objekttyp Adapter finden Sie in der *Natural for Ajax*-Dokumentation.

8 Map (Maske)

- Vorteile der Verwendung von Maps 56
- Map-Typen 56
- Maps erstellen 57
- Map-Verarbeitung starten/stoppen 57

Als Alternative zur dynamischen Spezifikation von Bildschirmmasken bietet das `INPUT`-Statement die Möglichkeit, vordefinierte Bildschirmmasken zu benutzen, und verwendet den Natural-Objekttyp „Map“ (Maske).

Vorteile der Verwendung von Maps

Die Benutzung von vordefinierten Bildschirmmasken im Gegensatz zu dynamischen Bildschirmmasken- Spezifikationen bietet verschiedene Vorteile wie z.B.:

- Klar strukturierte Anwendungen als Ergebnis einer konsequenten Trennung von Programm-Logik und Anzeige-Logik.
- Bildschirmmasken-Änderungen sind möglich, ohne Änderungen an den Hauptprogrammen vornehmen zu müssen.
- Die Sprache der Benutzerschnittstelle einer Anwendung kann leicht für internationale oder lokale Anforderungen angepasst werden.

Der Vorteil einer modularen Anwendungsstruktur mit Programmierobjekten, wie z.B. Maps, wird spätestens bei der Pflege von vorhandenen Natural-Anwendungen offenkundig.

Map-Typen

Maps (Bildschirmmasken) sind derjenige Teil einer Anwendung, den die Benutzer auf ihren Bildschirmen sehen.

Es gibt folgende Maskenarten:

- **Eingabemaske (Input Map)**
  Der Dialog mit dem Benutzer erfolgt über Eingabemasken.
- **Ausgabemaske (Output Map)**
  Wenn eine Anwendung einen Ausgabe-Report erzeugt, kann dieser Report mittels einer Ausgabemaske auf dem Bildschirm angezeigt werden.
- **Hilfemaske (Help Map)**
  Hilfemasken sind im Prinzip wie andere Maps, aber wenn sie als Hilfe zugewiesen werden, werden zusätzliche Prüfungen vorgenommen, um ihre Verwendbarkeit für Hilfe-Zwecke zu gewährleisten.

Der Objekttyp Map hat folgende Bestandteile:

- den Map-Hauptteil, in dem die Bildschirmmaske definiert ist und
- eine zugehörige **Parameter Data Area** (PDA), die als eine Art Schnittstelle Daten-Definitionen enthält, wie z.B. Name, Format, Länge jedes in einer spezifischen Map dargestellten Feldes.

Verwandte Themen:

- Informationen zu Auswahlboxen, die an Eingabefelder angehängt werden können, siehe Abschnitt *SB – Auswahlfenster (Selection Box)* in der *Statements*-Dokumentation (`INPUT`-Statement) und im Abschnitt *SB – Auswahlfenster (Selection Box)* in der *Parameter Reference*-Dokumentation.
- Informationen zu Split-Screen-Maps, in denen der obere Teil als Ausgabemaske und der untere Teil als Eingabemaske benutzt werden kann, entnehmen Sie dem Abschnitt *Split-Screen* in der *Statements*-Dokumentation (`INPUT`-Statement).

Maps erstellen

Maps und Helpmap-Layouts werden mit dem Map-Editor erstellt und bearbeitet. Die zugehörigen Datendefinitionen können aus einem anderen Natural-Objekt, zum Beispiel aus einer im Data-Area-Editor erstellten und gepflegten **Local Data Area** (LDA), gewählt werden.

In Abhängigkeit von der Plattform, auf der Natural betrieben wird, haben diese Editoren entweder eine zeichenorientierte Benutzeroberfläche oder eine grafische Benutzeroberfläche.

Verwandte Themen:

- Informationen zur Benutzung des Map-Editors entnehmen Sie dem Abschnitt *Map Editor* in der plattformspezifischen Natural *Editors* Dokumentation.
- Informationen zur Benutzung des Data-Area-Editors entnehmen Sie dem Abschnitt *Data Area Editor* in der plattformspezifischen *Editors*-Dokumentation.
- Informationen zur Eingabeverarbeitung mit dynamisch angegebenen Bildschirmmasken siehe Abschnitt *Syntax 1 – Dynamischer generierter Eingabeschirm* in der *Statements*-Dokumentation (`INPUT`-Statement).
- Informationen zur Eingabeverarbeitung mit einer mit dem Map-Editor erstellten Bildschirmmaske siehe Abschnitt *Syntax 2 – Verwendung einer vordefinierten Map* in der *Statements*-Dokumentation.

Map-Verarbeitung starten/stoppen

Eine Eingabemaske (Input Map) wird mit einem `INPUT USING MAP`-Statement aufgerufen.

Eine Ausgabemaske (Output Map) wird mit einem `WRITE USING MAP`-Statement aufgerufen.

Die Verarbeitung einer Map kann mit einem `ESCAPE ROUTINE`-Statement in einer Verarbeitungsregel (Processing Rule) gestoppt werden.

9 Helproutinen

- Helproutinen aufrufen 60
- Helproutinen spezifizieren 60
- Programmierhinweise für Helproutinen 61
- Parameter an Helproutinen übergeben 61
- Gleichheitszeichen-Option 62
- Array-Felder 63
- Hilfe als eingeblendetes Fenster 63

Helproutinen haben besondere Eigenschaften, die die Verarbeitung von Hilfe-Aufrufen erleichtern. Sie ermöglichen den Aufbau komplexer interaktiver Hilfe-Systeme. Helproutinen werden mit dem Programm-Editor erstellt.

Helproutinen aufrufen

In einer Natural-Anwendung fordern die Benutzer Hilfe an, indem Sie das Hilfe-Zeichen — standardmäßig ein Fragezeichen (?) — in einem Feld eingeben oder die Hilfe-Taste (normalerweise PF1) drücken. Dadurch rufen Sie eine Helproutine auf. Dabei gilt Folgendes:

- Das Hilfe-Zeichen ist nur einmal einzugeben.
- Das Hilfe-Zeichen muss das einzige geänderte Zeichen in der Eingabe-Zeichenkette sein.
- Das Hilfe-Zeichen muss das erste Zeichen in der Eingabe-Zeichenkette sein.

Auch in numerischen Feldern kann das Hilfe-Zeichen verwendet werden, wenn für das betreffende Feld eine Helproutine definiert ist. (Ungeachtet dessen überprüft Natural, ob es sich bei Eingaben in das Feld um gültige numerische Eingaben handelt.)

Die Hilfe-Taste kann - falls sie nicht bereits festgelegt ist - mit einem `SET KEY`-Statement bestimmt werden:

```
SET KEY PF1=HELP
```

Sie können eine Helproutine nur dann aufrufen, wenn sie in dem **Programm** oder der **Map**, von dem/der sie aufgerufen wird, spezifiziert ist.

Helproutinen spezifizieren

Eine Helproutine kann folgendermaßen angegeben werden:

- in einem Programm: auf Statement-Ebene und auf Feldebene,
- in einer Map: auf Maskenebene und auf Feldebene.

Wenn Sie Hilfe für ein Feld anfordern, dem keine Helproutine zugeordnet ist, oder wenn Sie Hilfe anfordern, ohne dass ein bestimmtes Feld referenziert wird, dann wird die auf der jeweiligen Statement- oder Maskenebene angegebene Helproutine aufgerufen.

Eine Helproutine kann auch über ein `REINPUT USING HELP`-Statement aufgerufen werden — entweder im Programm selbst oder in einer Verarbeitungsregel (Processing Rule). Falls das `REINPUT USING HELP`-Statement eine `MARK`-Option enthält, wird die Helproutine für das markierte Feld aufgerufen. Ist keine feldspezifische Helproutine angegeben, wird die Helproutine für die betreffende Map aufgerufen.

Ein REINPUT-Statement in einer Helproutine kann sich nur auf INPUT-Statements innerhalb derselben Helproutine beziehen.

Es gibt zwei Möglichkeiten, eine Helproutine anzugeben, entweder mit dem Session-Parameter HE in einem INPUT-Statement:

```
INPUT (HE='HELP2112')
```

oder im „Extended Field Editing“-Bereich des Map-Editors (wie in ***Maps erstellen*** und in der *Editors*-Dokumentation beschrieben).

Der Name einer Helproutine kann entweder eine alphanumerische Konstante sein oder eine alphanumerische Variable, die den Namen enthält. Falls es sich um eine Konstante handelt, muss der Name der Helproutine in Apostrophen angegeben werden.

Programmierhinweise für Helproutinen

Die Verarbeitung einer Helproutine kann mit einem ESCAPE ROUTINE-Statement gestoppt werden.

Bitte beachten Sie, dass ein END OF TRANSACTION- bzw. BACKOUT TRANSACTION-Statement in einer Helproutine die Transaktionslogik des Hauptprogramms beeinflusst.

Parameter an Helproutinen übergeben

Eine Helproutine kann auf die gerade aktive **Global Data Area** zugreifen (aber keine eigene Global Data Area haben). Außerdem kann sie eine eigene **Local Data Area** haben.

Darüber hinaus können Daten von der/an die Helproutine über Parameter übergeben werden. Eine Helproutine kann bis zu 20 explizite Parameter und einen impliziten Parameter haben. Die expliziten Parameter werden mit dem Operanden HE hinter dem Namen der Helproutine ausgegeben:

```
HE='MYHELP','001'
```

Der implizite Parameter ist das Feld, für das die Helproutine aufgerufen wurde:

```
INPUT #A (A5) (HE='YOURHELP','001')
```

wobei 001 der explizite Parameter ist und #A der implizite Parameter, also das Feld.

Dies wird im DEFINE DATA PARAMETER-Statement der Helproutine wie folgt definiert:

```
DEFINE DATA PARAMETER
1 #PARM1 (A3)          /* explicit parameter
1 #PARM2 (A5)          /* implicit parameter
END-DEFINE
```

Bitte beachten Sie, dass im obigen Beispiel der implizite Parameter #PARM2 auch weggelassen werden kann. Der implizite Parameter dient dazu, auf das Feld zuzugreifen, für das Hilfe angefordert wurde, sowie dazu, Daten von der Helproutine an das Feld zu übergeben. Es wäre beispielsweise denkbar, als Helproutine ein Rechenprogramm zu haben und das Rechenergebnis an das Feld zu übergeben.

Wenn Hilfe angefordert wird, so wird die Helproutine aufgerufen, bevor Daten vom Bildschirm an die Datenbereiche des Programms weitergegeben werden. Das bedeutet, dass Helproutinen nicht auf Daten zugreifen können, die während derselben Bildschirmtransaktion eingegeben wurden.

Nach Beenden der Helproutine werden auf dem Bildschirm die Feldwerte aktualisiert, die durch die Helproutine verändert wurden — mit Ausnahme der Felder, deren Inhalte bereits vorher vom Benutzer verändert worden waren, aber inklusive des Feldes, für das Hilfe angefordert wurde.

Ausnahme: Wenn das Feld, für das Hilfe angefordert wurde, durch dynamische Attribute (Session-Parameter DY) in mehrere Teile unterteilt wird und der Teil, in den das Fragezeichen eingegeben wurde, sich *nach* einem vom Benutzer veränderten Teil befindet, wird eine durch die Helproutine bewirkte Veränderung des Feldinhalts nicht wirksam.

Kontrollvariablen werden nach Beenden der Helproutine nicht noch einmal ausgewertet, selbst wenn sie innerhalb der Helproutine verändert wurden.

Gleichheitszeichen-Option

Es ist möglich, das Gleichheitszeichen (=) als expliziten Parameter anzugeben:

```
INPUT PERSONNEL-NUMBER (HE='HELPROUT',=)
```

Dieser Parameter wird dann als internes Feld (A65) verarbeitet, welches den Namen des Feldes bzw. der Map enthält. Die entsprechende Helproutine würde beispielsweise folgendermaßen beginnen:

```
DEFINE DATA PARAMETER
1 FNAME (A65)              /* contains 'PERSONNEL-NUMBER'
1 FVALUE (N8)              /* value of field (optional)
END-DEFINE
```

Diese Möglichkeit kann dazu eingesetzt werden, auf eine übergreifende Helproutine zuzugreifen, die den Feldnamen liest und feldspezifische Hilfe liefert, indem sie auf die Online-Dokumentation der Anwendung oder auf das Predict-Data-Dictionary zugreift.

Array-Felder

Ist das Feld, für das Hilfe aufgerufen wird, Teil eines **Arrays**, dann werden seine Indexangaben als implizite Parameter (1–3 je nach Rang ungeachtet der expliziten Parameter) angegeben.

Diese Parameter haben das Format I2.

```
INPUT A(*,*)  (HE='HELPROUT',=)
```

Die entsprechende Helproutine würde wie folgt beginnen:

```
DEFINE DATA PARAMETER
1 FNAME   (A65)             /* contains 'A'
1 FVALUE  (N8)              /* value of selected element
1 FINDEX1 (I2)              /* 1st dimension index
1 FINDEX2 (I2)              /* 2nd dimension index
END-DEFINE
...
```

Hilfe als eingeblendetes Fenster

Sie können die Größe eines Hilfe-Schirms so festlegen, dass sie kleiner ist als die Größe Ihres Bildschirms. In diesem Fall wird der Hilfe-Schirm als eingerahmtes Fenster auf dem Bildschirm eingeblendet:

```
*******************************************************************************
                           PERSONNEL INFORMATION
PLEASE ENTER NAME: ?__________________
PLEASE ENTER CITY: __________________
                   +---------------------------+
                   !                           !
                   ! Type in the name of an    !
                   ! employee in the first     !
                   ! field and press ENTER.    !
                   ! You will then receive     !
                   ! a list of all employees   !
                   ! of that name.             !
                   !                           !
                   ! For a list of employees   !
                   ! of a certain name who     !
                   ! live in a certain city,   !
                   ! type in a name in the     !
                   ! first field and a city    !
                   ! in the second field       !
                   ! and press ENTER.          !
*******************!                           !******************************
                   +---------------------------+
```

Innerhalb einer Helproutine haben Sie folgende Möglichkeiten, die Größe eines Fensters zu bestimmen:

- in einem `FORMAT`-Statement (z.B. um die Zeilen- und Seitenlänge anzugeben: `FORMAT PS=15 LS=30`)
- über ein `INPUT USING MAP`-Statement; in diesem Fall gilt die für die verwendete Map (in den **Map Settings**) festgelegte Größe
- durch ein `DEFINE WINDOW`-Statement; mit diesem Statement können Sie ein Fenster entweder explizit definieren oder dies Natural überlassen (Natural wird dann die Größe des Fensters je nach Inhalt festlegen).

Die Position des eingeblendeten Fensters wird automatisch in Abhängigkeit von der Position des Feldes, für das Hilfe angefordert wurde, bestimmt. Natural plaziert das Fenster automatisch möglichst nahe an das Feld, ohne es zu überdecken. Mit dem `DEFINE WINDOW`-Statement können Sie die Position des Fensters auch selbst bestimmen.

Weitere Informationen über Bildschirmfenster finden Sie beim `DEFINE WINDOW`-Statement in der *Statements*-Dokumentation und beim Terminalkommando `%W` in der *Terminalkommando*-Dokumentation.

10 Mehrfache Verwendung von Sourcecode — Copycode

- Copycode-Nutzung 66
- Copycode-Verarbeitung 66

Dieses Dokument beschreibt die Nutzung und Verarbeitung von Copycode.

Copycode-Nutzung

Copycode ist ein Stück Sourcecode, das mit einem `INCLUDE`-Statement in ein anderes Objekt eingefügt werden kann.

Wenn Sie einen Statement-Block haben, der in identischer Form in mehreren Objekten erscheinen soll, können Sie Copycode verwenden, anstatt den Statement-Block mehrmals zu kodieren. Dadurch reduziert sich der Kodieraufwand, und gleichzeitig ist sichergestellt, dass die Blöcke tatsächlich identisch sind.

Copycode-Verarbeitung

Der Copycode wird bei der Kompilierung eingefügt; d.h. die Sourcecode-Zeilen des Copycode werden nicht physisch in den Sourcecode des Objekts, das das `INCLUDE`-Statement enthält, eingefügt, sondern sie werden bei der Kompilierung berücksichtigt und sind so Bestandteil des resultierenden Objektmoduls.

Wenn Sie also den Sourcecode eines Copycode verändern, müssen Sie auch alle Objekte, in denen dieser Copycode verwendet wird, neu mit `STOW` kompilieren.

Achtung:

- Copycode kann nicht selbständig ausgeführt werden. Er kann nicht mit dem Systemkommando `STOW` in Objektform sondern nur in Sourceform mit dem Systemkommando `SAVE` gespeichert werden.
- Ein `END`-Statement kann nicht in einem Copycode platziert werden.

Weitere Informationen zu Copycode finden Sie bei der Beschreibung des `INCLUDE`-Statements in der *Statements*-Dokumentation.

11 Natural-Objekte dokumentieren — Text

- Verwendung des Natural-Objekttyps Text 68
- Text schreiben 68

Mit dem Natural-Objekttyp „Text“ können Sie Texte, zum Beispiel zu Dokumentationszwecken, erstellen.

Verwendung des Natural-Objekttyps Text

Sie können diesen Objekttyp dafür benutzen, eine Dokumentation für Natural-Objekte zu erstellen, die wesentlich umfangreicher sein kann, als es z.B. durch Kommentare innerhalb des Sourcecodes eines Objektes möglich wäre.

Der Objekttyp Text kann auch hilfreich sein, wenn Ihnen Predict nicht zur Dokumentation von Objekten zur Verfügung steht.

Text schreiben

Den Text schreiben Sie im Natural-Programm-Editor.

Der einzige Unterschied zur Programmerstellung liegt darin, dass keine Umsetzung von Klein- in Großbuchstaben vorgenommen wird, d.h. der Text, den Sie schreiben, bleibt unverändert.

Sie können einen beliebigen Text schreiben (eine Syntax-Prüfung gibt es nicht).

Textobjekte können nur in Sourceform (mit dem Systemkommando `SAVE`) gespeichert werden, aber nicht in Objektform (mit dem Systemkommando `STOW`).

Ein Textobjekt kann nicht mit `RUN` ausgeführt, sondern lediglich im Editor angezeigt werden.

12 Ereignisgesteuerte Anwendungen erstellen — Dialog

Dialoge werden in Verbindung mit ereignisgesteuerter Programmierung beim Erstellen von Natural-Anwendungen mit grafische Benutzeroberfläche (GUI) verwendet.

Weitere Informationen siehe ***Event-Driven Programming***.

13 Komponentenbasierte Anwendungen erstellen — Class

Der Natural-Objekttyp Class (Klasse) wird verwendet, um komponentenbasierte Anwendungen in einer Client/Server-Umgebung zu erstellen. Weitere Informationen siehe ***NaturalX*** im *Leitfaden zur Programmierung* und *Class Builder* in der *Editors*-Dokumentation.

14 Nicht-Natural-Dateien benutzen — Resource

- Verwendung von Resource-Objekten 74
- Shared Resources 74
- Private Resources 75

Dieses Kapitel behandelt folgende Themen:

Verwendung von Resource-Objekten

Natural unterscheidet zwei Arten von Resource-Objekten:

- **Shared Resources**
  Eine **Shared Resource** ist eine beliebige Nicht-Natural-Datei, die in einer Natural-Anwendung verwendet und im Natural-Library-System gepflegt wird.
- **Private Resources**
  Eine **Private Resource** ist eine Datei. die nur zu einem einzigen Natural-Obkelt zugeordnet ist und als Teil dieses Objekts betrachtet wird. Ein Objekt kann maximal eine Private-Resource-Datei haben. Gegenwärtig haben nur **Natural-Dialoge** Private Resources.

Sowohl diezu einer Library gehörenden Shared Resources als auch die Private Resources werden in einem Unterverzeichnis namens *..\RES* in dem Verzeichnis gepflegt, das die Natural-Library im Dateisystem darstellt.

Shared Resources

Eine Shared Resource ist eine beliebige Nicht-Natural-Datei, die in einer Natural-Anwendung verwendet und in einer Library im Natural-Library-System gepflegt wird. Eine Nicht-Natural-Datei, die als eine gemeinsam genutzte Resource genutzt werden soll, muss im Unterverzeichnis *..\RES* einer Natural-Library enthalten sein.

Beispiel für die Verwendung einer Shared Resource

Die Bitmap *MYPICTURE.BMP* soll in einem Bitmap-Control in einem Dialog namens MYDLG angezeigt werden, der in der Library MYLIB enthalten ist. Zuerst wird die Bitmap in die Libray MYLIB gestellt, indem man sie in das Verzeichnis *..\MYLIB\RES* verschiebt. Das folgende Stück Code aus dem Dialog MYDLG zeigt, wie sie dann dem Bitmap-Control zugewiesen wird:

```
DEFINE DATA LOCAL
01 #BM-1 HANDLE OF BITMAP
...
END-DEFINE
* (Creation of the Bitmap control omitted.)
...
#BM-1.BITMAP-FILE-NAME := "MYPICTURE.BMP" ...
```

Die Verwendung einer Bitmap als Shared Resource bietet die folgenden Vorteile:

- Die Datei kann in dem Natural-Dialog ohne Pfadname angegeben werden.
- Die Datei kann zusammen mit dem Objekt, das sie verwendet, in einer Natural-Library aufbewahrt werden.

Anmerkung: In früheren Natural-Versionen wurden Nicht-Natural-Dateien normalerweise in einem Verzeichnis aufbewahrt, das mit der Umgebungsvariablen `NATGUI_BMP` definiert wurde. Vorhandene Anwendungen, die diesen Ansatz verfolgen, funktionieren nicht mehr so wie zuvor, weil Natural immer in diesem Verzeichnis nach einer Shared-Resource-Datei sucht, wenn diese nicht in der aktuellen Library gefunden wurde.

Private Resources

Private Resources werden von Natural benutzt, um binäre Daten, die Teil von Natural-Objekten sind, zu speichern. Diese Dateien werden anhand der Dateinamenserweiterung *NR** erkannt, wobei "*" ein Zeichen ist, das vom Typ des Natural-Objekts abhängt. Natural führt die Pflege von Private Resources automatisch aus. Ein Natural-Objekt kann maximal eine Private-Resource-Datei haben.

Derzeit haben nur **Natural-Dialoge** Private-Resource-Dateien. Diese Datei wird benutzt, um darin die Konfiguration von ActiveX-Controls zu speichern, die in einem Dialog definiert und mit eigenen Property-Pages konfiguriert werden.

Informationen darüber, wie ein ActiveX-Control konfiguriert wird, finden Sie im Abschnitt *Attributes Windows for Dialogs and Dialog Elements*, *ActiveX Control Property Pages*.

Beispiele für Private Resources

Der Name der Private-Resource-Datei `MYDLG` ist *MYDLG.NR3*. Natural erstellt, ändert und löscht diese Datei bei Bedarf automatisch, wenn der Dialog erstellt, geändert, gelöscht usw. wird.

Die Private-Resource-Datei wird benutzt, um binäre Daten zu speichern, die in Bezug zu dem Dialog `MYDLG` stehen.

15 Felder definieren

Dieser Teil beschreibt, wie Sie die Felder definieren, die Sie in einem Programm verwenden möchten. Diese Felder können entweder Datenbankfelder oder benutzerdefinierte Felder sein.

- **Benutzung und Struktur des DEFINE DATA-Statements**
- **Benutzervariablen**
- **Function Call** (dieses Dokument ist nur in Englisch verfügbar)
- **Dynamische Variablen**
- **Dynamische und große Variablen benutzen**
- **Benutzerkonstanten**
- **Ausgangswerte (und das RESET-Statement)**
- **Felder redefinieren**
- **Arrays**
- **X-Arrays**

Bitte beachten Sie, dass dieses Kapitel sich auf die Hauptaspekte des `DEFINE DATA`-Statements beschränkt. Weitere Optionen bei diesem Statement sind in der Natural *Statements*-Dokumentation beschrieben.

Die Besonderheiten von Datenbankfeldern sind im Kapitel ***Datenbankzugriffe*** beschrieben. Im Prinzip gelten die dort für Adabas beschriebenen Funktionen und Beispiele auch für andere Datenbankverwaltungssysteme. Eventuell vorhandene Unterschiede sind in der betreffenden Datenbankschnittstellen-Dokumentation sowie in der *Statements*- bzw. der *Parameter Reference*-Dokumentation beschrieben.

16 Benutzung und Struktur des DEFINE DATA-Statements

- Felddefinitionen im DEFINE DATA-Statement ... 80
- Felder innerhalb eines DEFINE DATA-Statements definieren ... 81
- Felder in einer separaten Data Area definieren ... 81
- Struktur eines DEFINE DATA-Statements — Level-Nummern ... 82

Das erste Statement in einem im **Structured Mode** geschriebenen Natural-Programm muss immer ein `DEFINE DATA`-Statement sein. In diesem Statement definieren Sie alle Felder — Datenbankfelder wie Benutzervariablen — die in dem Programm verwendet werden sollen.

Informationen zur strukturellen Einrückung eines Source-Programms siehe Natural-Systemkommando `STRUCT`.

Felddefinitionen im DEFINE DATA-Statement

Alle im Programm zu verwendenden Felder *müssen* im `DEFINE DATA`-Statement definiert werden.

Es gibt zwei Möglichkeiten, die Felder zu definieren:

- Die Felder können innerhalb des `DEFINE DATA`-Statements selbst definiert werden (siehe **unten**).
- Die Felder können außerhalb des Programms in einer **Local Data Area oder einer Global Data Area** definiert werden, wobei das `DEFINE DATA`-Statement diese Data Area referenziert (siehe **unten**).

Felder, die von mehreren Programmen/Unterprogrammen benutzt werden, sollten in einer programmexternen Data Area definiert werden.

Im Hinblick auf eine klare Anwendungsstruktur empfiehlt es sich in der Regel, Felder in programmexternen Data Areas zu definieren.

Data Areas werden mit dem Data-Area-Editor erstellt und gepflegt, der in der *Editors*-Dokumentation beschrieben ist.

Im ersten der folgenden **Beispiele** sind die Felder innerhalb des `DEFINE DATA`-Statements im Programm selbst definiert. Im zweiten Beispiel sind die gleichen Felder in einer **Local Data Area** (LDA) definiert, und das `DEFINE DATA`-Statement enthält lediglich eine Referenz auf diese Data Area.

Felder innerhalb eines DEFINE DATA-Statements definieren

Das folgende Beispiel veranschaulicht, wie Felder innerhalb des Programms im DEFINE DATA-Statements selbst definiert werden können:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

Felder in einer separaten Data Area definieren

Das folgende Beispiel veranschaulicht, wie Felder außerhalb eines Programms in einer Local Data Area (LDA) definiert werden können:

Programm:

```
DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
...
```

Vom Programm referenzierte Local Data Area LDA39:

Diese LDA enthält eine View (Datensicht, siehe auch weiter **unten**) mit Namen VIEWEMP, in der die vom Programm verwendeten Felder definiert sind.

```
I T L Name                             F Leng Index/Init/EM/Name/Comment
- - - -------------------------------- - ---- --------------------------------
  V 1 VIEWEMP                               EMPLOYEES
    2 NAME                             A   20
    2 FIRST-NAME                       A   20
    2 PERSONNEL-ID                     A    8
    1 #VARI-A                          A   20
    1 #VARI-B                          N  3.2
```

```
    1 #VARI-C                             I     4
```

Struktur eines DEFINE DATA-Statements — Level-Nummern

Folgende Themen werden behandelt:

- Struktur und Gruppierung der Definitionen
- Level-Nummern in View-Definitionen
- Level-Nummern in Feldgruppen
- Level-Nummern in Redefinitionen

Struktur und Gruppierung der Definitionen

Level-Nummern werden innerhalb eines `DEFINE DATA`-Statements verwendet, um die Struktur und Gruppierung der Definitionen zu zeigen. Dies ist relevant bei

- **View-Definitionen**
- **Feldgruppen**
- **Redefinitionen**

Level-Nummern sind 1- oder 2-stellige Zahlen von 01 bis 99 (die vorangestellte Null kann weggelassen werden).

Im Allgemeinen sind Variablen-Definitionen auf Level 1.

Die Level-Angaben in View-Definitionen, Redefinitionen und Gruppen müssen lückenlos sein. Es dürfen keine Level-Nummern ausgelassen werden.

Level-Nummern in View-Definitionen

Wenn Sie einen View definieren, geben Sie den View-Namen auf Level 1 an und die Felder, aus denen der View besteht, auf Level 2. Näheres zu View-Definitionen finden Sie im Abschnitt ***Datenbankzugriffe***.

Beispiel für Level-Nummern in einer View-Definition:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
...
END-DEFINE
```

Level-Nummern in Feldgruppen

Mit der Definition von Gruppen ist es möglich, eine Reihe aufeinanderfolgender Felder auf einfache Weise zu referenzieren. Wenn Sie mehrere Felder unter einem gemeinsamen Gruppennamen definieren, können Sie diese Felder später im Programm referenzieren, indem Sie statt der Namen der einzelnen Felder lediglich den Gruppennamen angeben.

Der Gruppenname muss auf Level 1 definiert werden und die in der Gruppe enthaltenen Felder jeweils einen Level darunter.

Für Gruppennamen gelten die gleichen Namenskonventionen wie für Benutzervariablen. Siehe auch *Namenskonventionen für Benutzervariablen* in der Dokumentation *Natural Studio benutzen*.

Beispiel für Level-Nummern in einer Gruppe:

```
DEFINE DATA LOCAL
1 #FIELDA (N2.2)
1 #FIELDB (I4)
1 #GROUPA
  2 #FIELDC (A20)
  2 #FIELDD (A10)
  2 #FIELDE (N3.2)
1 #FIELDF (A2)
...
END-DEFINE
```

In diesem Beispiel sind die Felder `#FIELDC`, `#FIELDD` und `#FIELDE` unter dem gemeinsamen Gruppennamen `#GROUPA` definiert. Die anderen drei Felder sind nicht Teil der Gruppe. Bitte beachten Sie, dass `#GROUPA` nur als Gruppenname dient und selbst kein Feld ist (und daher auch keine Format-/Längendefinition hat).

Level-Nummern in Redefinitionen

Wenn Sie ein Feld redefinieren, muss die REDEFINE-Option auf dem gleichen Level sein wie die Definition des Feldes, das redefiniert wird; und die Felder, die sich aus der Redefinition ergeben, müssen einen Level darunter sein. Näheres zu Redefinitionen finden Sie unter ***Felder redefinieren***.

Beispiel für Level-Nummern in Redefinition:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF STAFFDDM
  2 BIRTH
  2 REDEFINE BIRTH
    3 #YEAR-OF-BIRTH  (N4)
    3 #MONTH-OF-BIRTH (N2)
    3 #DAY-OF-BIRTH   (N2)
1 #FIELDA (A20)
1 REDEFINE #FIELDA
  2 #SUBFIELD1 (N5)
  2 #SUBFIELD2 (A10)
  2 #SUBFIELD3 (N5)
...
END-DEFINE
```

In diesem Beispiel wird das Datenbankfeld BIRTH als drei Benutzervariablen redefiniert, und die Benutzervariable #FIELDA wird als drei andere Benutzervariablen redefiniert.

17 Benutzervariablen

- Definition von Benutzervariablen 86
- Datenbankfelder mit der (r)-Notation referenzieren 87
- Sourcecode-Zeilennummern umnummerieren 88
- Format und Länge von Benutzervariablen 89
- Spezielle Formate 91
- Index-Notation 93
- Datenbank-Array referenzieren 97
- Internen Zähler für ein Datenbank-Array referenzieren — C*-Notation 104
- Datenstrukturen qualifizieren 108
- Beispiele für Benutzervariablen 109

Von Ihnen selbst in einem Programm definierte Variablen werden als Benutzervariablen bezeichnet. Sie können Sie dazu verwenden, in einem Programm Werte oder Zwischenergebnisse zu speichern, die Sie später weiterverarbeiten oder ausgeben möchten.

Siehe auch *Namenskonventionen für Benutzervariablen* in der Dokumentation *Natural Studio benutzen*.

Definition von Benutzervariablen

Sie definieren eine Benutzervariable, indem Sie den Namen sowie das Format und die Länge der Variablen im `DEFINE DATA`-Statement angeben.

Sie definieren die Eigenschaften einer Variable mit der folgenden Notation:

```
(r,format-length/index)
```

Diese Notation folgt auf den Variablennamen, getrennt durch eine oder mehrere Leerstellen, als Option.

Zwischen den einzelnen Elementen der Notation sind keine Leerstellen zulässig.

Die einzelnen Elemente können erforderlichenfalls selektiv angegeben werden. Wenn sie aber zusammen benutzt werden, müssen Sie durch die oben angegebenen Zeichen voneinander getrennt werden.

Beispiel:

In diesem Beispiel ist eine Benutzervariable mit alphanumerischem Format und einer Länge von 10 Stellen unter dem Namen `#FIELD1` definiert.

```
DEFINE DATA LOCAL
1 #FIELD1 (A10)
...
END-DEFINE
```

Anmerkungen:

1. Im Strucutured Mode oder wenn ein Programm eine `DEFINE DATA LOCAL`-Klausel enthält, können Variablen nicht dynamisch in einem Statement definiert werden.
2. Dies gilt nicht für anwendungsunabhängige Variablen (AIVs). Siehe auch *Anwendungsunabhängige Variablen definieren*.

Datenbankfelder mit der (r)-Notation referenzieren

Ein Statement-Label oder die Sourcecode-Zeilennummer kann zum Referenzieren eines vorher eingesetzten Natural-Statements benutzt werden. Damit kann die Standard-Referenzierung von Natural überschrieben werden (wie für jedes einzelne Statement beschrieben, wo zutreffend), oder es wird zu Dokumentationszwecken verwendet. Siehe auch ***Schleifenverarbeitung***, Unterabschnitt ***Statements innerhalb eines Programms referenzieren***.

Folgende Themen werden behandelt:

- Standard-Referenzierung von Datenbankfeldern
- Referenzieren mit Statement-Labels
- Referenzieren mit Sourcecode-Zeilennummern

Standard-Referenzierung von Datenbankfeldern

Im Allgemeinen gilt Folgendes, wenn Sie keine Statement-Referenzierung spezifizieren:

- Standardmäßig wird die innerste aktive Datenbank-Schleife (`FIND`, `READ` oder `HISTOGRAM`) referenziert, in der das betreffende Datenbankfeld eingelesen wurde.
- Wenn das Feld in keiner aktiven Datenbank-Schleife eingelesen wurde, wird das letzte vorher verwendete `GET`-Statement (im Reporting Mode auch `FIND FIRST` oder `FIND UNIQUE`-Statement) referenziert, welches das Feld eingelesen hat.

Referenzieren mit Statement-Labels

Ein Natural-Statement, das bewirkt, dass eine Verarbeitungsschleife initiiert wird und/oder dass Datenelemente in der Datenbank aufgerufen werden, kann mit einem symbolischen Label zur nachfolgenden Referenzierung versehen werden.

Ein Label kann entweder in der Form "*label.*" vor dem referenzierenden Objekt oder in Klammern "(*label.*)" nach dem referenzierenden Objekt (aber nicht beide gleichzeitig) angegeben werden.

Die Namenskonventionen für Labels sind identisch mit denen für Variablen. Der Punkt nach dem Label-Namen dient zur Identifizierung des Eintrags als ein Label.

Beispiel:

```
...
 RD. READ PERSON-VIEW BY NAME STARTING FROM 'JONES'
   FD. FIND AUTO-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FD.)
         DISPLAY NAME (RD.) FIRST-NAME (RD.) MAKE (FD.)
    END-FIND
 END-READ
 ...
```

Referenzieren mit Sourcecode-Zeilennummern

Ein Statement kann auch referenziert werden, wenn dafür die Nummer der Sourcecode-Zeile, in der das Statement sich befindet, benutzt wird.

Alle vier Ziffern der Zeilennummer müssen angegeben werden (führende Nullen dürfen nicht weggelassen werden).

Beispiel:

```
...
0110 FIND EMPLOYEES-VIEW WITH NAME = 'SMITH'
0120   FIND VEHICLES-VIEW WITH MODEL = 'FORD'
0130     DISPLAY NAME (0110) MODEL (0120)
0140   END-FIND
0150 END-FIND
...
```

Sourcecode-Zeilennummern umnummerieren

Vierstellige, numerische Sourcecode-Zeilennummern, die ein Statement referenzieren (siehe *Notation zur Statement-Referenzierung — (r)*), werden auch umnummeriert, wenn das Natural-Sourceprogramm umnummeriert wird. Zur Unterstützung des Benutzers und zur Verbesserung der Programm-Lesbarkeit und -Fehlerbeseitigung werden alle in einem Statement, einer alphanumerischen Konstante oder in einem Kommentar auftretenden Sourcecode-Zeilennummern umnummeriert. Die Position der Sourcecode-Zeilennummer im Statement oder der alphanumerischen Konstante (Anfang, Mitte, Ende) spielt dabei keine Rolle.

Die folgende Zeichenfolge wird als eine gültige Referenz auf eine Sourcecode-Zeilennummer erkannt und umnummeriert (*nnnn* ist eine vierstellige Ziffer):

Zeichenfolge	Beispiel-Statement
`(nnnn)`	`ESCAPE BOTTOM (0150)`
`(nnnn /`	`DISPLAY ADDRESS-LINE(0010/1:5)`
`(nnnn,`	`DISPLAY NAME(0010,A10/1:5)`

Wenn der linken Klammer oder der vierstelligen Ziffer *nnnn* eine Leerstelle folgt, oder wenn der vierstelligen Ziffer *nnnn* ein Punkt folgt, gilt die Zeichenfolge nicht als eine gültige Referenz auf eine Sourcecode-Zeilennummer.

Um zu vermeiden, dass eine vierstellige, in einer alphanumerischen Konstante enthaltene Ziffer aus Zufall umnummeriert wird, sollte die Konstante aufgeteilt werden, und die unterschiedlichen Bestandteile sollten mittels eines Bindestriches miteinander zu einem Wert verkettet werden.

Beispiel:

```
Z := 'XXXX (1234,00) YYYY'
```

sollte ersetzt werden durch

```
Z := 'XXXX (1234' - ',00) YYYY'
```

Format und Länge von Benutzervariablen

Das Format und die Länge einer Benutzervariablen werden in Klammern hinter dem Namen der Variablen angegeben.

Variablen fester Länge können eine/s der folgenden Formate und Längen haben.

Zur Definition von Format und Länge bei dynamischen Variablen siehe ***Definition dynamischer Variablen***.

Format		Definierbare Länge	Interne Länge (in Bytes)
A	Alphanumerisch	1 - 1073741824 (1 GB)	1 - 1073741824
B	Binär	1 - 1073741824 (1 GB)	1 - 1073741824
C	Attribut-Zuweisung	-	2
D	Datum	-	4
F	Gleitkomma	4 oder 8	4 oder 8
I	Integer (Ganzzahl)	1 , 2 oder 4	1, 2 oder 4

Format		Definierbare Länge	Interne Länge (in Bytes)
L	Logisch	-	1
N	Numerisch (ungepackt)	1 - 29	1 - 29
P	Numerisch (gepackt)	1 - 29	1 - 15
T	Zeit	-	7
U	Unicode (UTF-16)	1 - 536870912 (0,5 GB)	2 - 1073741824

Die Länge kann nur angegeben werden, wenn das Format angegeben ist. Bei einigen Formaten braucht die Länge nicht explizit angegeben zu werden (wie in der Tabelle oben gezeigt).

Für mit Format N oder P definierte Felder können Sie die Dezimalstellen-Notation in der Form `nn.m` benutzen. `nn` stellt die Anzahl der Stellen vor dem Dezimalkomma dar, und `m` stellt die Anzahl der Stellen nach dem Dezimalkomma dar. Die Summe der Werte von `nn` und `m` darf 29 nicht überschreiten, und der Wert von of `m` darf 7 nicht überschreiten.

Anmerkungen:

1. Wenn eine Benutzervariable vom Format P mit einem `DISPLAY`-, `WRITE`- oder `INPUT`-Statement ausgegeben wird, wandelt Natural intern das Format P in das Format N für die Ausgabe um.
2. Wenn im Reporting Mode Format und Länge nicht für eine Benutzervariable angegeben werden, wird das/die Standard-Format/Länge N7 benutzt, es sei denn, diese Standard-Zuweisung wurde durch den Profil/Session-Parameter `FS` ausgeschaltet.

Für ein Datenbankfeld gilt das/die Format/Länge, wie für das Feld in der DDM definiert. (Im Reporting Mode ist es auch möglich, in einem Programm ein/e andere/s Format/Länge für ein Datenbankfeld zu definieren.)

Im Structured Mode kann Format und Länge nur in einer Data Area-Definition oder mit einem `DEFINE DATA`-Statement angegeben werden.

Beispiel für Format/Längen-Definition — Structured Mode:

```
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
1 #NEW-SALARY (N6.2)
END-DEFINE
...
FIND EMPLOY-VIEW ...
...
COMPUTE #NEW-SALARY = ...
...
```

Im Reporting Mode kann Format/Länge im Hauptteil des Programms definiert werden, wenn kein DEFINE DATA-Statement benutzt wird.

Beispiel für eine Format/Längen-Definition — Reporting Mode:

```
...
...
 FIND EMPLOYEES
... ... COMPUTE #NEW-SALARY(N6.2) = ...
...
```

Spezielle Formate

Zusätzlich zu den standardmäßigen Formaten alphanumerisch (A bzw. U) und numerisch (B, F, I, N, P) unterstützt Natural die folgenden speziellen Formate:

- Format C — Attribut-Kontrolle
- Formate D — Datum und T - Zeit
- Format L — Logisch
- Format Handle of Object

Format C — Attribut-Kontrolle

Eine mit dem Format C definierte Variable kann benutzt werden, um Attribute dynamisch einem in einem DISPLAY, INPUT-, PRINT-, PROCESS PAGE- oder WRITE-Statement benutzten Feld zuzuweisen.

Für eine Variable des Formats C kann keine Länge angegeben werden. Der Variable wird stets eine Länge von 2 Bytes von Natural zugewiesen.

Beispiel:

```
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A (N5)
END-DEFINE
...
MOVE (AD=I CD=RE) TO #ATTR
INPUT #A (CV=#ATTR)
...
```

Weitere Informationen siehe Session-Parameter CV.

Formate D — Datum und T - Zeit

Mit den Formaten D und T definierte Variablen können für die Datums- und Zeit-Arithmetik und -Anzeige benutzt werden. Das Format D kann nur Datums-Informationen enthalten. Das Format T kan sowohl Datums- als auch Zeit-Informationen enthalten; mit anderen Worten sind Datums-Informationen eine Untermenge der Zeit-Informationen. Die Zeit wird in Zehntelsekunden gemessen.

Für Variablen der Formate D und T kann keine Länge angegeben werden. Einer Variable mit Format D wird stets eine Länge von 4 Bytes (P6) und einer Variable mit Format T wird stets eine Länge von 7 Bytes (P12) von Natural zugewiesen. Wenn der Profilparameter MAXYEAR auf 9999 gesetzt ist, weist Natural einer Variablen des Formats D immer eine Länge von 4 Bytes (P7) zu, und einer Variablen des Formats T eine Länge von 7 Bytes (P13).

Beispiel:

```
DEFINE DATA LOCAL
1 #DAT1 (D)
END-DEFINE
*
MOVE *DATX TO #DAT1
ADD 7 TO #DAT1
WRITE '=' #DAT1
END
```

Weitere Informationen siehe Session-Parameter EM und Systemvariable *DATX und *TIMX.

Der Wert in einem Datumsfeld muss im Bereich vom 1. Januar 1582 bis 31. Dezember 2699 liegen.

Format L — Logisch

Eine mit Format L definierte Variable kann als ein logisches Bedingungskriterium benutzt werden. Sie kann den Wert TRUE (wahr) oder FALSE (falsch) annehmen.

Für eine Variable des Formats L kann keine Länge angegeben werden. Einer Variable des Formats L wird stets eine Länge von 1 Byte von Natural zugewiesen.

Beispiel:

```
DEFINE DATA LOCAL
1 #SWITCH(L)
END-DEFINE
MOVE TRUE TO #SWITCH
...
IF #SWITCH
   ...
   MOVE FALSE TO #SWITCH
ELSE
   ...
   MOVE TRUE TO #SWITCH
END-IF
```

Weitere Informationen zur logischen Wertedarstellung siehe Session-Parameter `EM`.

Format Handle of Object

Eine als `HANDLE OF OBJECT` definierte Variable kann als Objekt-Handle benutzt werden.

Weitere Informationen siehe ***NaturalX***.

Eine als `HANDLE OF` *`dialog-element-type`* definierte Variable kann als GUI-Handle benutzt werden.

Weitere Informationen zu GUI-Handles siehe ***HANDLE OF GUI*** im Abschnitt *Event-Driven Programming Techniques*.

Index-Notation

Eine Index-Notation wird für Felder benutzt, die ein Array darstellen.

Eine numerische Ganzzahl-Konstante oder Benutzervariable kann bei Index-Notationen eingesetzt werden. Eine Benutzervariable kann unter Benutzung von einem der folgenden Formate angegeben werden: N (numerisch), P (gepackt), I (Ganzzahl) oder B (binär), wobei das Format B nur mit einer Länge von kleiner oder gleich 4 benutzt werden kann.

Eine Systemvariable, Systemfunktion oder qualifizierte Variable kann nicht für Index-Notationen benutzt werden.

Array-Definition – Beispiele:

1. `#ARRAY (3)`

   Definiert ein eindimensionales Array mit drei Ausprägungen.

2. `FIELD (label.,A20/5)` oder `label.FIELD(A20/5)`

   Definiert ein Array von einem Datenbankfeld, das das durch *label* markierte Statement mit dem Format alphanumerisch, der Länge 20 und 5 Ausprägungen referenziert.

3. `#ARRAY (N7.2/1:5,10:12,1:4)`

   Definiert ein Array mit Format/Länge N7.2 und drei Array-Dimensionen mit 5 Ausprägungen (1 bis 5) in der ersten, 3 Ausprägungen (10 bis 12) in der zweiten und 4 Ausprägungen (1 bis 4) in der dritten Dimension.

4. `FIELD (label./i:i + 5)` oder `label.FIELD(i:i + 5)`

   Definiert ein Array von einem Datenbankfeld, das das durch *label.* markierte Statement referenziert.

   `FIELD` stellt ein multiples Feld oder ein Feld aus einer Periodengruppe dar, wobei *i* den Index für die relative Position innerhalb der Datenbank-Ausprägung angibt. Die Größe des Arrays innerhalb des Programms ist definiert als 6 Ausprägungen (i:i + 5). Der Datenbank-Index für die relative Position ist angegeben als eine Variable, um eine Positionierung des Programm-Arrays innerhalb der Ausprägungen des multiplen Feldes oder der Periodengruppe zu ermöglichen. Für eine erneute Positionierung von *i* muss über ein `GET`- oder `GET SAME`-Statement ein neuer Aufruf der Datenbank erfolgen.

Natural ermöglicht die Definition von Arrays, bei denen der Index nicht mit `1` anfangen muss. Zur Laufzeit überprüft Natural, ob in der Referenz angegebene Indexwerte nicht die maximale Größe der in der Definition spezifizierten Dimensionen überschreiten.

Anmerkungen:

1. Aus Kompatiblitätsgründen zu früheren Versionen von Natural kann ein Array-Bereich mittels eines Bindestrichs (-) anstatt eines Doppelpunkts (:) angegeben werden.
2. Eine Mischung aus beiden Notationen ist allerdings *nicht* zulässig.
3. Die Bindestrich-Notation ist nur zulässig im Reporting Mode (aber *nicht* in einem `DEFINE DATA`-Statement).

Der maximale Indexwert ist 1073741824. Die maximale Größe einer Data Area pro Programmierobjekt ist 1073741824 Bytes (1 GB).

Einfache arithmetische Ausdrücke können mittels der Operatoren „+“ und „-“ in Index-Referenzen verwendet werden. Wenn arithmetische Ausdrücke als Indizes benutzt werden, muss den Operatoren „+“ oder „-“ ein Leerzeichen vorausgehen und ihnen folgen.

Arrays in Gruppen-Strukturen werden von Natural Feld für Feld aufgelöst, und nicht Gruppen-Ausprägung für Gruppen-Ausprägung.

Beispiel für die Auflösung von Gruppen-Arrays:

```
DEFINE DATA LOCAL
 1 #GROUP (1:2)
   2 #FIELDA (A5/1:2)
   2 #FIELDB (A5)
 END-DEFINE
 ...
```

Wenn die oben definierte Gruppe in einem `WRITE`-Statement ausgegeben würde:

```
WRITE #GROUP (*)
```

würden die Ausprägungen in der folgenden Reihenfolge ausgegeben:

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(1) #FIELDB(2)
```

und *nicht*:

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDB(1) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(2)
```

Referenzieren von Arrays — Beispiele:

1. `#ARRAY (1)`

   Referenziert die erste Ausprägung eines eindimensionalen Arrays.
2. `#ARRAY (7:12)`

   Referenziert die siebente bis zwölfte Ausprägung eines eindimensionalen Arrays.
3. `#ARRAY (i + 5)`

   Referenziert die i+fünfte Ausprägung eines eindimensionalen Arrays.
4. `#ARRAY (5,3:7,1:4)`

   Es erfolgt eine Referenz innerhalb eines dreidimensionalen Arrays auf die Ausprägung 5 in der ersten Dimension, die Ausprägungen 3 bis 7 (5 Ausprägungen) in der zweiten Dimension und 1 bis 4 (4 Ausprägungen) in der dritten Dimension.

5. Stern-Notation (*) kann benutzt werden, um alle Ausprägungen innerhalb einer Dimension zu referenzieren:

```
DEFINE DATA LOCAL
1 #ARRAY1 (N5/1:4,1:4)
1 #ARRAY2 (N5/1:4,1:4)
END-DEFINE
...
ADD #ARRAY1 (2,*) TO #ARRAY2 (4,*)
...
```

Benutzung eines Schrägstrichs vor einer Array-Ausprägung

Wenn auf einen Variablennamen eine vierstellige Zahl in Klammern folgt, interpretiert Natural diese Zahl als eine Zeilennummer-Referenz auf ein Statement. Deshalb muss einer vierstelligen Array-Ausprägung ein Schrägstrich (/) vorausgehen, um anzuzeigen, dass es sich um eine Array-Ausprägung handelt, zum Beispiel:

```
#ARRAY(/1000)
```

und nicht:

```
#ARRAY(1000)
```

weil das Letztere als eine Referenz auf die Sourcecode-Zeile 1000 interpretiert würde.

Wenn ein Index-Variablenname als eine Format/Längen-Angabe fehlinterpretiert werden könnte, muss ein Schrägstrich (/) benutzt werden, um anzuzeigen, dass ein Index angegeben wird. Wenn z.B. die Ausprägung eines Arrays durch den Wert der Variable `N7` definiert ist, muss die Ausprägung angegeben werden als:

```
#ARRAY (/N7)
```

und nicht:

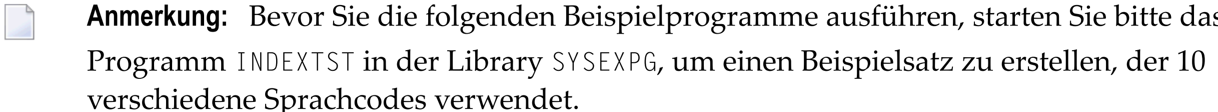
```
#ARRAY (N7)
```

weil das Letztere als die Definition eines 7 Bytes umfassenden numerischen Feldes fehlinterpretiert würde.

Datenbank-Array referenzieren

Folgende Themen werden behandelt:

- Multiple Felder und Periodengruppen-Felder referenzieren
- Mit Konstanten definierte Arrays referenzieren
- Mit Variablen definierte Arrays referenzieren
- Mehrfach definierte Arrays referenzieren



Anmerkung: Bevor Sie die folgenden Beispielprogramme ausführen, starten Sie bitte das Programm `INDEXTST` in der Library `SYSEXPG`, um einen Beispielsatz zu erstellen, der 10 verschiedene Sprachcodes verwendet.

Multiple Felder und Periodengruppen-Felder referenzieren

Ein multiples Feld oder ein Periodengruppen-Feld innerhalb eines Views bzw. einer DDM kann mittels verschiedener Index-Notationen definiert werden.

Zum Beispiel, der erste bis zehnte Wert und der Ite bis Ite+10 Wert desselben multiplen Feldes/Periodengruppen-Feldes eines Datenbanksatzes:

```
DEFINE DATA LOCAL
1 I (I2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 LANG (1:10)
  2 LANG (I:I+10)
END-DEFINE
```

oder:

```
RESET I (I2)
...
READ EMPLOYEES
OBTAIN LANG(1:10) LANG(I:I+10)
```

Anmerkungen:

1. Derselbe Index der unteren Grenze kann nur einmal pro Array benutzt werden (dies gilt für Konstanten-Indizes sowie für Variablen-Indizes).
2. Für eine Array-Definition unter Benutzung eines Variablen-Indexes muss die untere Grenze mittels der Variable selbst angegeben werden, und die obere Grenze muss mittels derselben Variable plus einer Konstante angegeben werden.

Mit Konstanten definierte Arrays referenzieren

Ein mit Konstanten definiertes Array kann entweder mittels Konstanten oder Variablen referenziert werden. Die obere Grenze des Array kann nicht überschritten werden. Die obere Grenze wird von Natural zur Kompilierungszeit geprüft, wenn eine Konstante benutzt wird.

Beispiel für den Reporting Mode:

```
** Example 'INDEX1R': Array definition with constants  (reporting mode)
************************************************************************
*
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (1:10)
  /*
  WRITE 'LANG(1:10):' LANG (1:10) //
  WRITE 'LANG(1)   :' LANG (1)    /  'LANG(5:9) :' LANG (5:9)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX1S': Array definition with constants (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 LANG (1:10)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1:10):' LANG (1:10) //
  WRITE 'LANG(1)   :' LANG (1)    /  'LANG(5:9) :' LANG (5:9)
END-READ
END
```

Wenn ein multiples Feld oder ein Periodengruppen-Feld mehrmals mittels Konstanten definiert wird und mittels Variablen referenziert werden soll, wird die folgende Syntax benutzt.

Beispiel für den Reporting Mode:

```
** Example 'INDEX2R': Array definition with constants (reporting mode)
**                   (multiple definition of same database field)
**********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:5)
  2 LANG (4:8)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  DISPLAY 'NAME'         NAME
          'LANGUAGE/1:3' LANG (1.1:3)
          'LANGUAGE/6:8' LANG (4.3:5)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX2S': Array definition with constants (structured mode)
**                   (multiple definition of same database field)
**********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:5)
  2 LANG (4:8)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  DISPLAY 'NAME'         NAME
          'LANGUAGE/1:3' LANG (1.1:3)
          'LANGUAGE/6:8' LANG (4.3:5)
END-READ
*
END
```

Mit Variablen definierte Arrays referenzieren

Mit Variablen definierte multiple Felder oder Periodengruppen-Felder in Arrays müssen mittels derselben Variable referenziert werden.

Beispiel für den Reporting Mode:

```
** Example 'INDEX3R': Array definition with variables (reporting mode)
************************************************************************
RESET I (I2)
*
I := 1
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (I:I+10)
  /*
  WRITE 'LANG(I)      :' LANG (I) /
        'LANG(I+5:I+7):' LANG (I+5:I+7)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX3S': Array definition with variables (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 I (I2)
*
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
END-DEFINE
*
I := 1
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(I)      :' LANG (I) /
        'LANG(I+5:I+7):' LANG (I+5:I+7)
END-READ
END
```

Wenn ein anderer Index benutzt werden soll, muss eine eindeutige Referenz auf die zuerst gefundene Definition des Arrays mit variablem Index erfolgen. Qualifizieren Sie dazu den Index-Ausdruck wie im Folgenden gezeigt.

Beispiel für den Reporting Mode:

```
** Example 'INDEX4R': Array definition with variables (reporting mode)
************************************************************************
RESET I (I2) J (I2)
*
I := 2
J := 3
*
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (I:I+10)
  /*
  WRITE 'LANG(I.J)  :' LANG (I.J) /
        'LANG(I.1:5):' LANG (I.1:5)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX4S': Array definition with variables (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 I (I2)
1 J (I2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
END-DEFINE
*
I := 2
J := 3
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(I.J)  :' LANG (I.J) /
        'LANG(I.1:5):' LANG (I.1:5)
END-READ
END
```

Der Ausdruck `I.` wird benutzt, um eine eindeutige Referenz auf die Array-Definition und „Positionen" zum ersten Wert innerhalb des Array-Lesebereichs (`LANG(I.1:5)`) zu erstellen.

Der aktuelle Inhalt von `I` zum Zeitpunkt des Datenbankzugriffs legt die Start-Ausprägung des Datenbank-Arrays fest.

Mehrfach definierte Arrays referenzieren

Für mehrfach definierte Arrays ist gewöhnlich eine Referenz mit Qualifizierung des Index-Ausdrucks erforderlich, um eine eindeutige Referenz auf den gewünschten Array-Bereich sicherzustellen.

Beispiel für den Reporting Mode:

```
** Example 'INDEX5R': Array definition with constants (reporting mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL                   /* For reporting mode programs
1 EMPLOY-VIEW VIEW OF EMPLOYEES     /* DEFINE DATA is recommended
  2 NAME                            /* to use multiple definitions
  2 CITY                            /* of same database field
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
I := 1
J := 2
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
LOOP
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX5S': Array definition with constants (structured mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
```

```
*
I := 1
J := 2
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
END-READ
END
```

Eine ähnliche Syntax wird auch benutzt, wenn multiple Felder oder Periodengruppen-Felder unter Benutzung von Index-Variablen definiert werden.

Beispiel für den Reporting Mode:

```
** Example 'INDEX6R': Array definition with variables (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES    /* For reporting mode programs
  2 NAME                           /* DEFINE DATA is recommended
  2 CITY                           /* to use multiple definitions
  2 LANG (I:I+10)                  /* of same database field
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
  WRITE 'LANG(I.I)    :' LANG (I.I) /
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
LOOP
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX6S': Array definition with variables (structured mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
  WRITE 'LANG(I.I)    :' LANG (I.I) /
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
END-READ
END
```

Internen Zähler für ein Datenbank-Array referenzieren — C*-Notation

Es ist manchmal erforderlich, ein multiples Feld und/oder eine Periodengruppe zu referenzieren, ohne zu wissen, wie viele Werte/Ausprägungen in einem gegebenen Datensatz vorhanden sind. Adabas unterhält einen internen Zähler der Anzahl für die Werte jedes einzelnen multiplen Feldes und die Anzahl der Ausprägungen jeder einzelnen Periodengruppe. Dieser Zähler kann unter Angabe von `C*` unmittelbar vor dem Feldnamen referenziert werden.

Anmerkung in Bezug auf andere Datenbanken als Adabas:

Tamino	Bei XML-Datenbanken kann die C* Notation nicht verwendet werden.
SQL	Bei SQL-Datenbanken kann die C* Notation nicht verwendet werden.

Das/Die zum Deklarieren eines C*-Feldes zulässige explizite Format und Länge ist entweder

- Ganzzahl-Format Integer (I) mit einer Länge von 2 Bytes (I2) oder 4 Bytes (I4),
- numerisch (N) oder gepackt (P) mit einer ganzzahligen Ziffer (aber ohne Dezimalstellen), zum Beispiel (N3).

Wenn kein explizites Format und keine explizite Länge angegeben wird, ist Format/Länge N3 die Voreinstellung.

Beispiele:

C*LANG	gibt den Zähler der Anzahl der Werte für das multiple Feld LANG zurück.
C*INCOME	gibt den Zähler der Anzahl der Ausprägungen für die Periodengruppe INCOME zurück.
C*BONUS(1)	gibt den Zähler der Anzahl der Werte für das multiple Feld BONUS in der Periodengruppen-Ausprägung 1 zurück (wenn man davon ausgeht, dass BONUS ein multiples Fel.d innerhalb einer Periodengruppe ist.)

Beispielprogramm mit C*-Notation bei Variablen:

```
** Example 'CNOTX01': C* Notation
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*INCOME
  2 INCOME
    3 SALARY  (1:5)
    3 C*BONUS (1:2)
    3 BONUS   (1:2,1:2)
  2 C*LANG
  2 LANG      (1:2)
*
1 #I (N1)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  WRITE NOTITLE 'NAME:' NAME /
       'NUMBER OF LANGUAGES SPOKEN:' C*LANG 5X
       'LANGUAGE 1:' LANG (1) 5X
```

```
       'LANGUAGE 2:' LANG (2)
  /*
  WRITE 'SALARY DATA:'
  FOR #I FROM 1 TO C*INCOME
    WRITE 'SALARY' #I SALARY (1.#I)
  END-FOR
  /*
  WRITE 'THIS YEAR BONUS:' C*BONUS(1)  BONUS (1,1) BONUS (1,2)
      / 'LAST YEAR BONUS:' C*BONUS(2)  BONUS (2,1) BONUS (2,2)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `CNOTX01`:

```

NAME: SENKO
NUMBER OF LANGUAGES SPOKEN:    1     LANGUAGE 1: ENG     LANGUAGE 2:
SALARY DATA:
SALARY  1      36225
SALARY  2      29900
SALARY  3      28100
SALARY  4      26600
SALARY  5      25200
THIS YEAR BONUS:    0          0          0
LAST YEAR BONUS:    0          0          0

NAME: CANALE
NUMBER OF LANGUAGES SPOKEN:    2     LANGUAGE 1: FRE     LANGUAGE 2: ENG
SALARY DATA:
SALARY  1     202285
THIS YEAR BONUS:    1      23000          0
LAST YEAR BONUS:    0          0          0
```

C*-Notation bei multiplen Felder innerhalb von Periodengruppen

Für ein multiples Feld innerhalb einer Periodengruppe können Sie auch eine mit `C*`-Notation versehene Variable mit einer Indexbereichsangabe definieren.

Bei den folgenden Beispielen wird das multiple Feld `BONUS` benutzt, das Teil der Periodengruppe `INCOME` ist. Alle drei Beispiele liefern dasselbe Ergebnis.

Beispiel 1 — Reporting Mode:

```
** Example 'CNOTX02': C* Notation (multiple-value fields)
************************************************************************
*
LIMIT 2
READ EMPLOYEES BY CITY
  OBTAIN C*BONUS (1:3)
         BONUS   (1:3,1:3)
  /*
  DISPLAY NAME C*BONUS (1:3) BONUS (1:3,1:3)
LOOP
*
END
```

Beispiel 2 — Structured Mode:

```
** Example 'CNOTX03': C* Notation (multiple-value fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 INCOME    (1:3)
    3 C*BONUS
    3 BONUS   (1:3)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  DISPLAY NAME C*BONUS (1:3) BONUS (1:3,1:3)
END-READ
*
END
```

Beispiel 3 — Structured Mode:

```
** Example 'CNOTX04': C* Notation (multiple-value fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*BONUS  (1:3)
  2 INCOME   (1:3)
    3 BONUS  (1:3)
END-DEFINE
```

```
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  DISPLAY NAME C*BONUS (*) BONUS (*,*)
END-READ
*
END
```

Vorsicht: Da der Adabas-Formatpuffer keine Bereiche für Zählerfelder zulässt, werden sie als einzelne Felder generiert; deshalb kann ein `C*`-Indexbereich für ein großes Array zu einem Adabas Formatpuffer-Überlauf führen.

Datenstrukturen qualifizieren

Um ein Feld beim Referenzieren zu identifizieren, können Sie das Feld qualifizieren; d.h. vor dem Feldnamen geben Sie den Namen des Level-1-Datenelementes ein, in dem das Feld sich befindet, und einen Punkt.

Ist ein Feld nicht eindeutig über den Namen zu identifizieren (z.B. wenn derselbe Feldname in mehreren Gruppen/Views benutzt wird), müssen Sie es beim Referenzieren qualifizieren.

Die Kombination von Level-1-Datenelement und Feldnamen muss eindeutig sein.

Beispiel:

```
DEFINE DATA LOCAL
1 FULL-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
1 OUTPUT-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
END-DEFINE
...
MOVE FULL-NAME.LAST-NAME TO OUTPUT-NAME.LAST-NAME
...
```

Der Kennzeichner muss ein Level-1-Datenelement sein.

Beispiel:

```
DEFINE DATA LOCAL
1 GROUP1
  2 SUB-GROUP
    3 FIELD1 (A15)
    3 FIELD2 (A15)
END-DEFINE
...
MOVE 'ABC' TO GROUP1.FIELD1
...
```

Datenbankfeld qualifizieren:

Benutzen Sie denselben Namen für eine Benutzervariable und ein Datenbankfeld (was Sie nicht tun sollten), müssen Sie das Datenbankfeld qualifizieren, wenn Sie es referenzieren wollen.

Vorsicht: Wenn Sie ein Datenbankfeld mit gleichem Namen wie eine Benutzervariable nicht qualifizieren, wird die Benutzervariable referenziert.

Beispiele für Benutzervariablen

```
DEFINE DATA LOCAL
1 #A1 (A10)       /* Alphanumeric, 10 positions.
1 #A2 (B4)        /* Binary, 4 positions.
1 #A3 (P4)        /* Packed numeric, 4 positions and 1 sign position.
1 #A4 (N7.2)      /* Unpacked numeric,
                  /* 7 positions before and 2 after decimal point.
1 #A5 (N7.)       /* Invalid definition!!!
1 #A6 (P7.2)      /* Packed numeric, 7 positions before and 2 after decimal point
                  /* and 1 sign position.
1 #INT1 (I1)      /* Integer, 1 byte.
1 #INT2 (I2)      /* Integer, 2 bytes.
1 #INT3 (I3)      /* Invalid definition!!!
1 #INT4 (I4)      /* Integer, 4 bytes.
1 #INT5 (I5)      /* Invalid definition!!!
1 #FLT4 (F4)      /* Floating point, 4 bytes.
1 #FLT8 (F8)      /* Floating point, 8 bytes.
1 #FLT2 (F2)      /* Invalid definition!!!
1 #DATE (D)       /* Date (internal format/length P6).
1 #TIME (T)       /* Time (internal format/length P12).
1 #SWITCH (L)     /* Logical, 1 byte (TRUE or FALSE).
                  /*
END-DEFINE
```

18 Function Call

- Calling User-Defined Functions 112
- Restrictions 113
- Syntax Description 113

```
call-name(<([prototype-cast][intermediate-result-definition])[parameter][,[parameter]]...>)
```

Related topics: DEFINE FUNCTION | DEFINE PROTOTYPE

Calling User-Defined Functions

Function calls can be used to call **user-defined functions** which are defined inside special objects of type **function**.

There are different ways to call a function:

- Symbolic Function Call
- Function Call Using a Variable

Symbolic Function Call

When using the symbolic function call, the user specifies exactly the name of the function to be executed at runtime.

If only a symbolic function call is specified in the Natural source, the corresponding Natural function definition is retrieved automatically, unless a suitable prototype definition has been specified before. The corresponding name of the object, which contains the Natural function definition, is retrieved according to the symbolic logical function name. This is done using the link records of the *FILEDIR.SAG* file. In this case, the corresponding function definition must have been stowed before the link record can be generated for the first time.

This feature causes that all parameter definitions of a Natural function call are always checked for valid format/length definitions.

Function Call Using a Variable

In a function call using a variable, the name of the desired function definition is stored inside an alphanumeric variable. At runtime, Natural jumps into the corresponding function definition, the name of which has been stored inside the variable.

In order to identify these two kinds of function calls, a corresponding prototype definition must be specified. Additionally, the prototype may contain the whole signature of the function definition. If no signature has been given, the function call must contain a `PT` clause in order to specify the missing parts of the signature. Therefore, the `VARIABLE` keyword of such a prototype specified inside the `PT` clause has no effect. For variable function calls, there must be a valid prototype with the same name as the alphanumeric variable containing the function name.

If no prototype can be assigned to the function call, a special *prototype-cast* is necessary in order to define the return format/length at compilation time. The *prototype-cast* and the parameter list must be enclosed in pointy brackets and parentheses, as displayed in the syntax diagram.

If you want to use the variable method, you must define a prototype with the same name as the *variable-name* using the keyword VARIABLE.

Example:

```
DEFINE PROTOTYPE VARIABLE variable-name
```

Anmerkung: You can only use a function call when the operand involved cannot be modified. However, if a function call is used in an INPUT statement, the return value will be displayed as an „output only“ field (AD=O).

Restrictions

Function calls are *not* allowed in the following situations:

- in a DEFINE DATA statement;
- in a database access or update statement (READ, FIND, SELECT, UPDATE, STORE, etc.);
- in an AT BREAK or IF BREAK statement;
- as an argument of the system functions AVER, COUNT, MAX, MIN, NAVER, NCOUNT, NMIN, OLD, SUM, TOTAL;
- as index notation.

Syntax Description

A function call may consist of the following syntax elements:

- call-name
- prototype-cast
- intermediate-result-definition

- parameter

call-name

```
{ function-name           }
{ prototype-variable-name }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
prototype-variable-name		S	A			A	U											yes	no

Syntax Element Description:

function-name	The *function-name* clause is the symbolic function name. The corresponding function definition is defined in a certain function object file.
prototype-variable-name	The *prototype-variable-name* is the name of the variable containing the real name of the function which is to be called. An alphanumeric or Unicode variable with the same name must have already been defined.

prototype-cast

```
PT= { prototype-name          }
    { prototype-variable-name }
```

The *prototype-cast* must be used for function calls where no signature is specified in the corresponding function prototype (for example, signature clause of prototype definition is defined as `UNKNOWN`).

intermediate-result-definition

```
    { format-length [/array-definition]                        }
IR= {                                                          }
    {       { A                      }                         }
    { (     { U  [/array-definition] }   )      DYNAMIC        }
    {       { B                      }                         }
```

This clause enables you to specify the *format-length/array-definition* of the return value for a function call without using an explicit or implicit prototype definition, that is, it enables the explicit specification of an intermediate result.

If, in addition, a prototype is valid for the function call, it is checked that the *format-length*/*array-definition* of the return value of the function definition is move-compatible to the intermediate result. If this is not the case, an error will be raised. The intermediate result is taken for the return value.

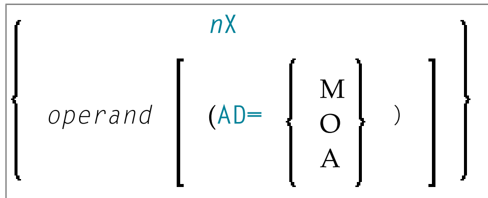
Alternatively, arrays are possible as return values, that is, array definitions may be specified as intermediate results. With an *array-definition*, you define the lower and upper bound of a dimension in an array definition. See *Array Dimension Definition* in the *Statements* documentation.

format-length	The format and length of the field. For information on format/length definition of user-defined variables, see ***Format and Length of User-Defined Variables***.
A, B or U	Data type: Alphanumeric, Binary or Unicode for dynamic variables.
array-definition	With an *array-definition*, you define the lower and upper bounds of the dimensions in an array definition. See *Array Dimension Definition* in the *Statements* documentation.
DYNAMIC	A field may be defined as DYNAMIC. For more information on processing dynamic variables, see ***Introduction to Dynamic Variables and Fields***.

parameter

Each parameter may be an operand when calling the function. If a parameter is defined with the keyword OPTIONAL in the subprogram's DEFINE DATA PARAMETER statement, the corresponding operand values may be omitted in the function call. In this case, use the *n*X notation (where *n* is a whole integer greater than or equal to 1) or just omit this argument.

You can specify the session parameter AD for each argument.



Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	yes	yes

For an example of the proper usage of this function call, see the example in the description of the `DEFINE PROTOTYPE` statement.

<table>
<tr><td rowspan="1">nX</td><td colspan="2">Parameters to be Skipped:

With the notation nX you can specify that the next n parameters are to be skipped (for example, 1X to skip the next parameter, or 3X to skip the next three parameters); this means that for the next n parameters no values are passed to the subprogram.

A parameter that is to be skipped must be defined with the keyword OPTIONAL in the subprogram's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.</td></tr>
<tr><td rowspan="5">AD=</td><td colspan="2">Attribute Definition:

If operand is a variable, you can mark it in one of the following ways:</td></tr>
<tr><td>AD=O</td><td>Non-modifiable, see session parameter AD=O.

Anmerkung: Internally, AD=O is processed in the same way as BY VALUE (see the section parameter-data-definition in the description of the DEFINE DATA statement).</td></tr>
<tr><td>AD=M</td><td>Modifiable, see session parameter AD=M.

This is the default setting.</td></tr>
<tr><td>AD=A</td><td>Input only, see session parameter AD=A.</td></tr>
<tr><td colspan="2">If operand is a constant, AD cannot be explicitly specified. For constants, AD=O always applies.</td></tr>
</table>

19 Dynamische Variablen

- Sinn und Zweck dynamischer Variablen 118
- Definition dynamischer Variablen 119
- Zurzeit für eine dynamische Variable benutzter Wertespeicher 119
- Größenbeschränkungsprüfung 120
- Hauptspeicherplatz für eine dynamische Variable zuweisen/freigeben 120

Sinn und Zweck dynamischer Variablen

Insofern als die maximale Größe der großen Datenstrukturen (zum Beispiel Bilder, Audiosignale, Videos) zur Anwendungsentwicklungszeit nicht genau bekannt sein kann, bietet Natural zusätzlich die Definition alphanumerischer und binärer Variablen mit dem Attribut `DYNAMIC`.

Der Wertebereich von Variablen, die mit diesem Attribut definiert sind, wird zur Ausführungszeit dynamisch erweitert, wenn es notwendig wird (zum Beispiel bei einer Zuweisungsoperation: `#picture1 := #picture2`). Dies bedeutet, dass große binäre und alphanumerische Datenstrukturen in Natural verarbeitet werden können, ohne dass Sie eine Beschränkung zur Entwicklungszeit definieren müssen.

Die Zuweisung von dynamischen Variablen zur Ausführungszeit unterliegt natürlich den Beschränkungen des verfügbaren Hauptspeichers. Wenn die Zuweisung dynamischer Variablen dazu führt, dass eine Meldung wegen unzureichenden Hauptspeichers vom zugrundeliegenden Betriebssystem zurückgegeben wird, kann das `ON ERROR`-Statement benutzt werden, um diese Fehlerbedingung abzufangen, sonst wird eine Fehlermeldung von Natural zurückgegeben.

Die Natural-Systemvariable `*LENGTH` kann benutzt werden, um die Länge (in Code- Einheiten) des Wertespeichers zu erhalten, der zur Zeit für eine gegebene dynamische Variable benutzt wird. Für Format A und B ist die Größe einer Code-Einheit 1 Byte. Für Format U ist die Größe einer Code-Einheit 2 Bytes (UTF–16). Natural setzt `*LENGTH` automatisch auf die Länge des Source-Operanden bei Zuweisungen, in denen die dynamische Variable betroffen ist. `*LENGTH(`*`field`*`)` gibt deshalb die aktuell für ein/e dynamische/s Natural-Feld oder eine Variable in Code-Einheiten benutzte Größe zurück.

Wenn der Speicherplatz für die dynamische Variable nicht mehr erforderlich ist, kann das `REDUCE`- oder `RESIZE`-Statement benutzt werden, um den für die dynamische Variable benutzten Speicherplatz auf Null (oder eine andere gewünschte Größe) zu reduzieren. Wenn die obere Grenze der Hauptspeicher-Benutzung für eine spezifische dynamische Variable bekannt ist, kann das `EXPAND`-Statement benutzt werden, um den für die dynamische Variable benutzten Speicherplatz auf diese bestimmte Größe zu setzen.

Große Variablen für alphanumerische und binäre Daten basieren auf den bekannten Natural-Formaten A und B. Die Beschränkungen von 253 Bytes für Format A und 126 für Format B sind nicht mehr gültig. Die neue Größenbeschränkung ist 1 GB. Diese großen statischen Variablen und Felder werden genauso verarbeitet wie traditionelle alphanumerische und binäre Variablen und Felder in Bezug auf Definition, Redefinition, Wertespeicherzuweisung, Konvertierungen, Referenzierungen in Statements, usw. Alle Regeln zu alphanumerischen und binären Formaten gelten für diese großen Formate.

Soll eine dynamische Variable initialisiert werden, sollte das `MOVE ALL UNTIL`-Statement für diesen Zweck benutzt werden.

Definition dynamischer Variablen

Da die wirkliche Größe von großen alphanumerischen und binären Datenstrukturen zur Anwendungsentwicklungszeit nicht genau bekannt sein mag, kann die Definition *dynamischer* Variablen des Formats A, B oder U zur Verwaltung dieser Strukturen benutzt werden. Die dynamische Zuweisung und Erweiterung (Neuzuweisung) großer Variablen ist gegenüber der Anwendungsprogrammierungslogik transparent. Dynamische Variablen werden ohne Längenangabe definiert. Der Hauptspeicher wird entweder implizit zur Ausführungszeit zugewiesen, wenn die dynamische Variable als ein Ziel-Operand benutzt wird, oder explizit mit einem `EXPAND`- oder `RESIZE`-Statement.

Dynamische Variablen können nur in einem `DEFINE DATA`-Statement mittels der folgenden Syntax definiert werden:

```
level variable-name(format) DYNAMIC
```

Dabei ist:

- *level* die Stufennummer
- *variable-name* der Name der großen Variablen
- *format* das Format der Variablen (A, U oder B)
- DYNAMIC das Schlüsselwort, durch das die Variable als dynamsiche Variable definiert wird.

Die folgenden Beschränkungen gelten für eine dynamische Variable:

- Eine Redefinition einer dynamischen Variable ist nicht zulässig.
- Eine dynamische Variable darf nicht in einer `REDEFINE`-Klausel enthalten sein.

Zurzeit für eine dynamische Variable benutzter Wertespeicher

Die Größe (in Code-Einheiten) des zurzeit benutzten Wertespeichers einer dynamischen Variable kann aus der Systemvariable `*LENGTH` übernommen werden. `*LENGTH` wird bei Zuweisungen automatisch auf die (verwendete) Länge des Source-Operanden gesetzt.

Vorsicht: Aus Performance-Gründen kann der zur Aufnahme des Wertes der dynamischen Variablen zugewiesene Speicherbereich größer sein als der Wert von `*LENGTH` (benutzte Größe steht dem Entwickler zur Verfügung). Sie sollten sich nicht auf den Speicherplatz verlassen, der über die benutzte Länge (wie durch `*LENGTH` angegeben) hinaus zugewiesen wird. Dieser Platz kann jederzeit freigegeben werden, auch wenn auf die betreffende dynamische Variable nicht zugegriffen wird. Es ist für den Natural-Programmierer nicht

möglich, Informationen über die aktuell zugewiesene Größe zu erhalten. Diese Größe ist ein interner Wert.

Die Systemvariable `*LENGTH(field)` gibt die benutzte Länge eines dynamischen Natural-Feldes oder einer solchen Variable in Code-Einheiten zurück. `*LENGTH` kann nur benutzt werden, um die aktuell verwendete Länge für dynamische Variablen zu erhalten.

Größenbeschränkungsprüfung

Profilparameter USIZE

Bei dynamischen Variablen ist eine Größenbeschränkungsprüfung während der Kompilierung nicht möglich, da bei dynamischen Variablen keine Länge definiert ist. Der Profilparameter `USIZE` gibt die Größe des Benutzerpuffers im virtuellen Speicher an. Der Benutzerpuffer enthält alle Daten, die von Natural dynamisch zugewiesen wurden. Wenn eine dynamische Variable während der Ausführung zugewiesen oder erweitert wird und die `USIZE`-Grenze überschritten wird, wird eine Fehlermeldung ausgegeben.

Hauptspeicherplatz für eine dynamische Variable zuweisen/freigeben

Die Statements `EXPAND`, `REDUCE` und `RESIZE` werden benutzt, um Hauptspeicherplatz für eine dynamische Variable explizit zuzuweisen und freizugeben.

Syntax:

```
EXPAND [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
REDUCE [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
RESIZE [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
```

dabei ist *operand1* eine dynamische Variable und *operand2* ein nicht-negativer, numerischer Längenwert ist.

EXPAND

Das `EXPAND`-Statement wird benutzt, um den zurzeit *zugewiesenen* Speicherplatz der dynamischen Variable (*operand1*) auf die angegebene Größe (*operand2*) zu erweitern.

Die aktuell benutzte Größe (siehe Natural-Systemvariable `*LENGTH` weiter **oben**) für die dynamische Variable wird nicht geändert.

Wenn die angegebene Größe (*operand2*) kleiner ist als die Größe des aktuell zugewiesenen Speicherplatzes der dynamischen Variable, wird das Statement ignoriert.

REDUCE

Das `REDUCE`-Statement wird benutzt, um die Größe des zurzeit *zugewiesenen* Speicherplatzes der dynamischen Variable (*operand1*) auf die angegebene Größe (*operand2*) zu reduzieren.

Der über die angegebene Größe (*operand2*) hinausgehende, für die dynamische Variable (*operand1*) zugewiesene Speicherplatz kann jederzeit freigegeben werden, wenn das Statement ausgeführt wird, oder auch später.

Wenn die zurzeit *benutzte* Speichergröße (siehe Natural-Systemvariable `*LENGTH` weiter **oben**) für die dynamische Variable größer ist als die angegebene Speichergröße (*operand2*), wird `*LENGTH` dieser dynamischen Variable auf die angegebene Größe gesetzt. Der Inhalt der Variablen wird abgeschnitten, aber nicht geändert.

Wenn die angegebene Größe größer ist als der zurzeit zugewiesene Speicherplatz der dynamischen Variable, wird das Statement ignoriert.

RESIZE

Das `RESIZE`-Statement passt die Größe des zurzeit zugewiesenen Speicherplatzes der dynamischen Variable (*operand1*) an die angegebene Größe (*operand2*) an.

Wenn die angegebene Größe der dynamischen Variable kleiner ist als die benutzte Größe (siehe Natural-Systemvariable `*LENGTH` weiter **above**), wird die benutzte Größe dementsprechend reduziert.

Wenn die *angegebene* Größe größer ist als die Größe des zurzeit zugewiesenen Speicherplatzes der dynamischen Variable, wird die Größe des zugewiesenen Speicherplatzes der dynamischen Variable erhöht. Die aktuell benutzte Größe (siehe `*LENGTH`) der dynamischen Variable ist davon nicht betroffen und bleibt unverändert.

Wenn die angegebene Größe identisch ist mit der Größe des zurzeit zugewiesenen Speicherplatzes der dynamischen Variable, hat die Ausführung des `RESIZE`-Statements keine Auswirkungen.

20 Dynamische und große Variablen benutzen

- Allgemeine Informationen zu dynamischen Variablen 124
- Zuweisungen mit dynamischen Variablen 125
- Initialisierung dynamischer Variablen 127
- String-Manipulation mit dynamischen alphanumerischen Variablen 127
- Logische Bedingungen (LCC) bei dynamischen Variablen 129
- AT/IF-BREAK dynamischer Kontrollfelder 130
- Parameter-Übertragung mit dynamischen Variablen 131
- Arbeitsdateizugriff bei großen und dynamischen Variablen 134
- DDM-Generierung und Editieren von Spalten mit variabler Länge 135
- Zugriff auf große Datenbankobjekte 137
- Performance-Aspekte bei dynamischen Variablen 138
- Ausgabe von dynamische Variablen 140
- Dynamische X-Arrays 140

Allgemeine Informationen zu dynamischen Variablen

Im Allgemeinen kann eine dynamische alphanumerische Variable immer dann benutzt werden, wenn ein alphanumerisches Feld zulässig ist. Ein dynamisches binäres Feld kann immer dann benutzt werden, wenn ein binäres Feld erlaubt ist. Ein dynamisches Unicode-Feld kann überall dort verwendet werden, wo ein Unicode-Feld erlaubt ist.

Ausnahme:

Dynamische Variablen sind nicht zulässig beim `SORT`-Statement. Um dynamische Variablen in einem `DISPLAY`-, `WRITE`-, `PRINT`-, R`REINPUT`- oder `INPUT`-Statement zu benutzen, müssen Sie entweder den Session-Parameter `AL` oder `EM` benutzen, um die Länge der Variable zu definieren.

Die benutzte Länge (siehe Natural-Systemvariable `*LENGTH`, ***Aktuell für eine dynamische Variable benutzter Wertespeicher***) und die Größe des zugewiesenen Speicherplatzes der dynamischen Variablen sind gleich Null, bis auf die Variable als Ziel-Operand zum ersten Mal zugegriffen wird. Aufgrund von Zuweisungen oder anderen Operationen können dynamische Variablen zuerst zugewiesen oder auf die exakte Größe des Source-Operanden erweitert werden (neu zugewiesen).

Die Größe einer dynamischen Variable kann mit den folgenden Statements erweitert werden, wenn sie als ein änderbarer Operand (Ziel-Operand) in den folgenden Statements benutzt wird:

ASSIGN	*operand1* (Ziel-Operand in einer Zuweisung).
CALLNAT	Siehe *Parameter-Übertragung bei dynamischen Variablen* (außer wenn `AD=O`, oder wenn `BY VALUE` in der entsprechenden PDA vorhanden ist)
COMPRESS	*operand2*, siehe *Verarbeitung*.
EXAMINE	*operand1* in der `DELETE REPLACE`-Klausel.
MOVE	*operand2* (Ziel-Operand), siehe *Funktion*.
PERFORM	Außer wenn `AD=O`, oder wenn `BY VALUE` in der betreffenden PDA vorhanden ist.
READ WORK FILE	*operand1* und *operand2*, siehe *Verarbeitung großer und dynamischer Variablen*.
SEPARATE	*operand4*.
SELECT (SQL)	Parameter in der `INTO`-Klausel.
SEND METHOD	*operand3* (außer wenn `AD=O`).

Zur Zeit gibt es die folgende Beschränkung in Bezug auf die Verwendung von großen Variablen:

CALL	Parameter-Größe kleiner als 64 KB pro Parameter (keine Beschränkung für `CALL` mit INTERFACE4-Option).

In den folgenden Abschnitten wird die Benutzung der dynamischen Variablen detaillierter und mit Beispielen erörtert.

Zuweisungen mit dynamischen Variablen

Im Allgemeinen wird eine Zuweisung in der zurzeit benutzten Länge des Source-Operanden durchgeführt (siehe Natural-Systemvariable `*LENGTH`). Wenn der Ziel-Operand eine dynamische Variable ist, wird seine zurzeit zugewiesene Größe gegebenenfalls erweitert, um den Source-Operanden ohne Abschneiden zu verschieben.

Beispiel:

```
#MYDYNTEXT1 := OPERAND
MOVE OPERAND TO #MYDYNTEXT1
/* #MYDYNTEXT1 IS AUTOMATICALLY EXTENDED UNTIL THE SOURCE OPERAND CAN BE COPIED
```

`MOVE ALL` bzw. `MOVE ALL UNTIL` mit dynamischen Ziel-Operanden wird wie folgt definiert:

- `MOVE ALL` verschiebt den Source-Operanden wiederholt zum Ziel-Operanden, bis die benutzte Länge (`*LENGTH`) des Ziel-Operanden erreicht ist. `*LENGTH` wird nicht geändert. Wenn `*LENGTH` Null ist, wird das Statement ignoriert.
- `MOVE ALL` *`operand1`* `TO` *`operand2`* `UNTIL operand3` verschiebt *`operand1`* wiederholt zu *`operand2`*, bis die in *`operand3`* angegebene Länge erreicht ist. Wenn *`operand3`* größer als `*LENGTH(`*`operand2`*`)` ist, wird *`operand2`* erweitert, und `*LENGTH(`*`operand2`*`)` wird auf *`operand3`* gesetzt. Wenn *`operand3`* kleiner als `*LENGTH(`*`operand2`*`)` ist, wird die benutzte Länge auf *`operand3`* reduziert. Wenn *`operand3`* gleich `*LENGTH(`*`operand2`*`)` ist, entspricht das Verhalten dem bei `MOVE ALL`.

Beispiel:

```
#MYDYNTEXT1 := 'ABCDEFGHIJKLMNO'           /* *LENGTH(#MYDYNTEXT1) = 15
MOVE ALL 'AB' TO #MYDYNTEXT1               /* CONTENT OF #MYDYNTEXT1 =
'ABABABABABABABA';
                                           /* *LENGTH IS STILL 15
MOVE ALL 'CD' TO #MYDYNTEXT1 UNTIL 6       /* CONTENT OF #MYDYNTEXT1 = 'CDCDCD';
                                           /* *LENGTH = 6
MOVE ALL 'EF' TO #MYDYNTEXT1 UNTIL 10     /* CONTENT OF #MYDYNTEXT1 = 'EFEFEFEFEF';
                                           /* *LENGTH = 10
```

MOVE JUSTIFIED wird zur Kompilierungszeit zurückgewiesen, wenn der Ziel-Operand eine dynamische Variable ist.

MOVE SUBSTR und MOVE TO SUBSTR sind zulässig. MOVE SUBSTR führt zu einem Laufzeitfehler, wenn ein Substring hinter der benutzten Länge einer dynamischen Variable (*LENGTH) referenziert wird. MOVE TO SUBSTR führt zu einem Laufzeitfehler, wenn eine Substring-Position hinter *LENGTH + 1 referenziert wird, weil dies zu einer undefinierten Lücke im Inhalt der dynamischen Variable führen würde. Wenn der Ziel-Operand von MOVE TO SUBSTR erweitert werden sollte (zum Beispiel, wenn der zweite Operand auf *LENGTH+1 gesetzt wird), ist der dritte Operand zwingend.

Gültige Syntax:

```
#OP2 := *LENGTH(#MYDYNTEXT1)
MOVE SUBSTR (#MYDYNTEXT1, #OP2) TO OPERAND              /* MOVE LAST CHARACTER
TO OPERAND
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2, #lEN_OPERAND)   /* CONCATENATE OPERAND
TO #MYDYNTEXT1
```

Ungültige Syntax:

```
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE SUBSTR (#MYDYNTEXT1, #OP2, 10) TO OPERAND          /* LEADS TO RUNTIME ERROR;
UNDEFINED SUB-STRING
#OP2 := *LENGTH(#MYDYNTEXT1 + 10)
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2, #EN_OPERAND)    /* LEADS TO RUNTIME ERROR;
UNDEFINED GAP
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2)               /* LEADS TO RUNTIME ERROR;
UNDEFINED LENGTH
```

Zuweisungskompatibilität

Beispiel:

```
#MYDYNTEXT1   := #MYSTATICVAR1
#MYSTATICVAR1 := #MYDYNTEXT2
```

Wenn der Source-Operand eine statische Variable ist, wird die benutzte Länge des dynamischen Ziel-Operanden (*LENGTH(#MYDYNTEXT1)) auf die Format-Länge der statischen Variablen gesetzt, und der Source-Wert wird einschließlich nachfolgender Leerzeichen (alphanumerische und Unicode-Felder) oder binärer Nullen (für binäre Felder) in diese Länge kopiert.

Wenn der Ziel-Operand statisch und der Source-Operand dynamisch ist, wird die dynamische Variable in ihre zurzeit benutzte Länge kopiert. Wenn diese Länge kleiner als die Format-Länge

der statischen Variable ist, wird der Rest mit Leerzeichen (für alphanumerische und Unicode-Felder) oder binären Nullen (für binäre Felder) aufgefüllt, sonst wird der Wert abgeschnitten. Wenn die aktuell benutzte Länge der dynamischen Variable Null (0) ist, wird der statische Ziel-Operand mit Leerzeichen (für alphanumerische und Unicode-Felder) oder binären Nullen (für binäre Felder) aufgefüllt.

Initialisierung dynamischer Variablen

Dynamische Variablen können mit einem `RESET`-Statement mit Leerzeichen (alphanumerische und Unicode-Felder) oder Nullen (binäre Felder) bis zur aktuell benutzten Länge (= `*LENGTH`) initialisiert werden. `*LENGTH` wird nicht geändert.

Beispiel:

```
DEFINE DATA LOCAL
  1 #MYDYNTEXT1   (A)  DYNAMIC
END-DEFINE
#MYDYNTEXT1 := 'SHORT TEXT'
WRITE *LENGTH(#MYDYNTEXT1)          /* USED LENGTH = 10
RESET #MYDYNTEXT1                   /* USED LENGTH = 10, VALUE = 10 BLANKS
```

Um eine dynamische Variable mit einem angegebenen Wert in einer angegebenen Länge zu initialisieren, kann das `MOVE ALL UNTIL`-Statement benutzt werden.

Beispiel:

```
MOVE ALL 'Y' TO #MYDYNTEXT1 UNTIL 15     /* #MYDYNTEXT1 CONTAINS 15 'Y'S, USED
LENGTH = 15
```

String-Manipulation mit dynamischen alphanumerischen Variablen

Wenn ein änderbarer Operand eine dynamische Variable ist, wird ihre zurzeit zugewiesene Länge möglicherweise erhöht, um die Operation ohne Abschneidung oder Fehlermeldung auszuführen. Dies gilt für die Verkettung (`COMPRESS`) und Trennung von dynamischen alphanumerischen Variablen (`SEPARATE`).

Beispiel:

```
** Example 'DYNAMX01': Dynamic variables (with COMPRESS and SEPARATE)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1 (A)   DYNAMIC
1 #TEXT       (A20)
1 #DYN1       (A)   DYNAMIC
1 #DYN2       (A)   DYNAMIC
1 #DYN3       (A)   DYNAMIC
END-DEFINE
*
MOVE ' HELLO WORLD ' TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading and trailing blanks
*
MOVE ' HELLO WORLD ' TO #TEXT
*
MOVE #TEXT TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with whole variable length of #TEXT
*
COMPRESS #TEXT INTO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading blanks of #TEXT
*
*
#MYDYNTEXT1 := 'HERE COMES THE SUN'
SEPARATE #MYDYNTEXT1 INTO #DYN1 #DYN2 #DYN3 IGNORE
*
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
WRITE #DYN1 (AL=25) 'with length' *LENGTH (#DYN1)
WRITE #DYN2 (AL=25) 'with length' *LENGTH (#DYN2)
WRITE #DYN3 (AL=25) 'with length' *LENGTH (#DYN3)
/*  #DYN1, #DYN2, #DYN3 are automatically extended or reduced
*
EXAMINE #MYDYNTEXT1 FOR 'SUN' REPLACE 'MOON'
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  #MYDYNTEXT1 is automatically extended or reduced
*
END
```

Anmerkung: Im Falle von nicht-dynamischen Variablen kann eine Fehlermeldung zurückgegeben werden.

Logische Bedingungen (LCC) bei dynamischen Variablen

Im Allgemeinen erfolgt eine Lese-Operation (z.B. ein Vergleich) bei einer dynamischen Variablen mit ihrer zurzeit benutzten Größe. Dynamische Variablen werden wie statische Variablen verarbeitet, wenn sie in einem Lese-Kontext (nicht änderbar) benutzt werden.

Beispiel:

```
IF  #MYDYNTEXT1  =  #MYDYNTEXT2  OR  #MYDYNTEXT1 =  "**"  THEN ...
IF  #MYDYNTEXT1  <  #MYDYNTEXT2  OR  #MYDYNTEXT1 <  "**"  THEN ...
IF  #MYDYNTEXT1  >  #MYDYNTEXT2  OR  #MYDYNTEXT1 >  "**"  THEN ...
```

Auch im Falle von nachfolgenden Leerzeichen oder führenden Nullen zeigen dynamische Variablen ein entsprechendes Verhalten.

Für dynamische Variablen ist der alphanumerische Wert `AA` gleich `AA`, und der binäre Wert `00003031` ist gleich `3031`.

Führende Nullen für alphanumerische und Unicode-Variablen oder führende binäre Nullen für binäre Variablen werden bei statischen und dynamischen Variablen gleich behandelt. Zum Beispiel werden alphanumerische Variable, die die Werte `AA` und `AA ` (d.h. `AA` mit nachfolgendem Leerzeichen) enthalten, als gleich angesehen. Binäre Variablen, die beispielsweise die Werte H'0000031' und H'3031' enthalten, werden ebenso als gleich angesehen. Wenn ein Vergleichsergebnis nur im Falle einer exakten Kopie wahr (`TRUE`) sein sollte, müssen die benutzten Längen der dynamischen Variablen außerdem miteinander verglichen werden. Wenn eine Variable eine exakte Kopie der anderen ist, sind ihre benutzten Längen auch gleich.

Beispiel:

```
#MYDYNTEXT1  :=  'HELLO'                                  /* USED LENGTH IS 5
#MYDYNTEXT2  :=  'HELLO     '                             /* USED LENGTH IS 10
IF   #MYDYNTEXT1  =  #MYDYNTEXT2   THEN  ...              /* TRUE
IF   #MYDYNTEXT1  =  #MYDYNTEXT2 AND
     *LENGTH(#MYDYNTEXT1) = *LENGTH(#MYDYNTEXT2)  THEN  ...  /* FALSE
```

Zwei dynamische Variablen werden Position für Position miteinander verglichen (von links nach rechts bei alphanumerischen Variablen und von rechts nach links bei binären Variablen) bis zum Minimum ihrer benutzten Längen. Die erste Position, an der die Variablen nicht gleich sind, legt fest, ob die erste oder zweite Variable größer, gleich oder kleiner als die andere ist. Die Variablen sind gleich, wenn sie bis zum Minimum ihrer benutzten Längen gleich sind, und der Rest der längeren Variable nur Leerzeichen (bei alphanumerischen dynamischen Variablen) oder binäre Nullen (bei dynamischen binären Variablen) enthält. Um zwei dynamische Unicode-Variablen miteinander zu vergleichen, werden aus beiden Werten die führenden Nullen entfernt, bevor zum

Vergleich der beiden resultierenden Werte der ICU-Collation-Algoritmus benutzt wird. Siehe auch *Logical Condition Criteria* in der *Unicode and Code Page Support*-Dokumentation.

Beispiel:

```
#MYDYNTEXT1  :=  'HELLO1'                     /* USED LENGTH IS 6
#MYDYNTEXT2  :=  'HELLO2'                     /* USED LENGTH IS 10
IF #MYDYNTEXT1  <  #MYDYNTEXT2 THEN  ...      /* TRUE
#MYDYNTEXT2  :=  'HALLO'
IF #MYDYNTEXT1  >  #MYDYNTEXT2 THEN  ...      /* TRUE
```

Vergleichskompatibilität

Vergleiche zwischen dynamischen und statischen Variablen sind gleichwertig mit Vergleichen zwischen dynamischen Variablen. Die Format-Länge der statischen Variable wird als ihre benutzte Länge interpretiert.

Beispiel:

```
#MYSTATTEXT1 :=  'HELLO'                       /* FORMAT LENGTH OF MYSTATTEXT1 IS
A20
#MYDYNTEXT1  :=  'HELLO'                       /* USED LENGTH IS 5
IF  #MYSTATTEXT1  =  #MYDYNTEXT1  THEN  ...    /* TRUE
IF  #MYSTATTEXT1  >  #MYDYNTEXT1  THEN  ...    /* FALSE
```

AT/IF-BREAK dynamischer Kontrollfelder

Der Vergleich des Gruppenwechsel-Kontrollfeldes mit seinem alten Wert wird Position für Position von links nach rechts durchgeführt. Wenn der alte und der neue Wert der dynamischen Variable jeweils unterschiedliche Längen hat, dann wird zu Vergleichszwecken der Wert mit der kürzeren Länge nach rechts aufgefüllt (mit Leerzeichen für alphanumerisch und Unicode (dynamische Werte oder binäre Nullen für binäre Werte).

Im Falle eines alphanumerischen oder Unicode-Gruppenwechselkontrollfeldes sind nachfolgende Leerzeichen nicht bedeutend für den Vergleich, d.h. nachfolgende Leerzeichen bedeuten keine Änderung des Wertes, und es tritt kein Gruppenwechsel auf.

Im Falle eines binären Gruppenwechselkontrollfeldes sind nachfolgende binäre Nullen nicht bedeutend für den Vergleich, d.h. nachfolgende binäre Nullen bedeuten keine Änderung des Wertes, und es findet kein Gruppenwechsel statt.

Parameter-Übertragung mit dynamischen Variablen

Dynamische Variablen können als Parameter an aufgerufene Programmobjekte (`CALLNAT`, `PERFORM`) übergeben werden. Aufruf über eine Referenz (Call-by-Reference) ist möglich, weil der Wertespeicher einer dynamischen Variable zusammenhängend ist. Aufruf über Wert (Call-by-Value) führt zu einer Zuweisung der Variablen-Definition des Aufrufenden als Source-Operand und der Parameter-Definition als Ziel-Operand. Bei Aufruf über Wert (Ergebnis) (Call-by-Value (Result)) ist es umgekehrt.

Bei Aufruf über eine Referenz (Call-by-Reference) müssen beide Definitionen dynamisch (DYNAMIC) sein. Wenn nur eine von ihnen dynamisch ist, tritt ein Laufzeitfehler auf. Im Falle von Call-by-Value (Result), d.h. Aufruf über Wert (Ergebnis) sind alle Kombinationen möglich. Die folgende Tabelle veranschaulicht die gültigen Kombinationen:

Call-By-Reference

Caller	Parameter	
	Statisch	Dynamisch
Statisch	Ja	Nein
Dynamisch	Nein	Ja

Die Formate der dynamischen Variablen A oder B müssen miteinander im Einklang sein.

Call-by-Value (Result)

Caller	Parameter	
	Statisch	Dynamisch
Statisch	Ja	Ja
Dynamisch	Ja	Ja

Anmerkung: Im Falle von statischen/dynamischen oder dynamischen/statischen Definitionen können gemäß der Datenübertragungsregeln der betreffenden Zuweisungen Werte abgeschnitten werden.

Beispiel 1:

```
** Example 'DYNAMX02': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX03' USING #MYTEXT
*
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
*
END
```

Subprogramm `DYNAMX03`:

```
** Example 'DYNAMX03': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA PARAMETER
1 #MYPARM (A) DYNAMIC  BY VALUE RESULT
END-DEFINE
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 6
#MYPARM := '1234567'                    /* *LENGTH(#MYPARM) = 7
#MYPARM := '12345678'                   /* *LENGTH(#MYPARM) = 8
EXPAND DYNAMIC VARIABLE #MYPARM TO 10   /* 10 bytes are allocated
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 8
*
/* content of #MYPARM is moved back to #MYTEXT
/* used length of #MYTEXT = 8
*
END
```

Beispiel 2:

```
** Example 'DYNAMX04': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX05' USING #MYTEXT
```

```
*
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
                                /* at least 10 bytes are
                                /* allocated (extended in DYNAMX05)
*
END
```

Subprogramm DYNAMX05:

```
** Example 'DYNAMX05': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA PARAMETER
1 #MYPARM (A) DYNAMIC
END-DEFINE
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 6
#MYPARM := '1234567'                    /* *LENGTH(#MYPARM) = 7
#MYPARM := '12345678'                   /* *LENGTH(#MYPARM) = 8
EXPAND DYNAMIC VARIABLE #MYPARM TO 10   /* 10 bytes are allocated
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 8
*
END
```

3GL-Progamm aufrufen

Dynamische und große Variablen können mit dem CALL-Statement sinnvoll benutzt werden, wenn die Option INTERFACE4 verwendet wird. Der Einsatz dieser Option führt zu einer Schnittstelle zum 3GL-Programm mit einer unterschiedlichen Parameter-Struktur.

Dazu sind einige geringfügige Änderungen an dem 3GL-Programm erforderlich, jedoch bringt es folgende signifikanten Vorteile im Vergleich zu der früheren FINFO-Struktur:

- Keine Einschränkung bei der Anzahl der übergebenen Parameter (früher begrenzt auf 40)
- Keine Einschränkung bei der Datengröße eines Parameters (früher auf 64 KB pro Parameter begrenzt)
- Die Parameterinformationen können vollständig an das 3GL-Programm übergeben werden (einschließlich Array-Informationen). Exportierte Funktionen sind verfügbar, die einen sicheren Zugriff auf die Parameterdaten ermöglichen (früher musste darauf geachtet werden, nicht den Speicher in Natural zu überschreiben).

Weitere Informationen zur FINFO-Struktur siehe CALL INTERFACE4-Statement.

Bevor Sie ein 3GL-Programm mit dynamischen Parametern aufrufen, ist es wichtig sicherzustellen, dass die erforderliche Puffergröße zugewiesen wird. Dies kann explizit über das EXPAND-Statement erfolgen.

Wenn ein initialisierter Puffer erforderlich ist, kann die dynamische Variable mittels des `MOVE ALL UNTIL`-Statements auf den Ausgangswert und auf die erforderliche Größe gesetzt werden. Natural stellt eine Reihe von Funktionen zur Verfügung, die es dem 3GL-Programm ermöglichen, Informationen über dynamische Parameter zu erhalten und die Länge zu ändern, wenn Parameterdaten zurückgeschrieben werden.

Beispiel:

```
MOVE ALL ' ' TO  #MYDYNTEXT1 UNTIL 10000
  /* a buffer of length 10000 is allocated
  /* #MYDYNTEXT1 is initialized with blanks
  /* and *LENGTH(#MYDYNTEXT1) = 10000
CALL INTERFACE4  'MYPROG'  USING  #MYDYNTEXT1
WRITE *LENGTH(#MYDYNTEXT1)
  /* *LENGTH(#MYDYNTEXT1) may have changed in the 3GL program
```

Eine ausführlichere Beschreibung finden Sie beim `CALL`-Statement in der *Statements*-Dokumentation.

Arbeitsdateizugriff bei großen und dynamischen Variablen

Folgende Themen werden behandelt:

- PORTABLE und UNFORMATTED
- ASCII, ASCII-COMPRESSED und SAG
- Besondere Bedingungen für TRANSFER und ENTIRE CONNECTION

PORTABLE und UNFORMATTED

Große und dynamische Variablen können mit den beiden Arbeitsdateitypen PORTABLE and UNFORMATTED in Arbeitsdateien geschrieben werden oder aus Arbeitsdateien gelesen werden. Bei diesen Arbeitsdateitypen gibt es keine Größenbeschränkung für dynamische Variablen. Große Variablen dürfen die maximale Feld-/Datensatzlänge von 32766 Bytes jedoch nicht überschreiten.

Beim Arbeitsdateityp PORTABLE wird die Feldinformation in der Arbeitsdatei gespeichert. Bei dynamischen Variablen wird die Größe während des `READ` geändert, wenn die Feldgröße im Datensatz von der aktuellen Größe abweicht.

Der Arbeitsdateityp UNFORMATTED kann zum Beispiel dazu genutzt werden, ein Video aus einer Datenbank auszulesen und es in einer Datei abzulegen, die mit anderen Dienstprogrammen direkt abspielbar ist. Mit dem `WRITE WORK`-Statement werden die Felder mit ihrer Byte-Länge in die Datei geschrieben. Alle Datentypen (`DYNAMIC` oder auch nicht) werden gleich behandelt. Es werden keine strukturellen Informationen eingefügt. Da Natural einen Puffermechanismus benutzt, werden die Daten erst nach einem `CLOSE WORK` komplett geschrieben. Dies ist dann besonders

wichtig, wenn die Datei mit einem anderen Dienstprogramm verarbeitet werden soll während Natural aktiv ist.

Mit dem `READ WORK`-Statement werden Felder mit einer festen Länge mit ihrer gesamten Länge gelesen. Wenn das Ende der Datei erreicht wird, wird der Rest des aktuellen Feldes mit Leerzeichen aufgefüllt. Die nachfolgenden Felder bleiben unverändert. Im Falle von `DYNAMIC`-Datentypen wird der komplette Rest der Datei gelesen, außer wenn sie 1073741824 Bytes überschreitet. Wenn das Ende der Datei erreicht wird, bleiben die übrigen Felder (Variablen) unverändert (normales Natural-Verhalten).

ASCII, ASCII-COMPRESSED und SAG

Die Arbeitsdateitypen ASCII, ASCII-COMPRESSED und SAG (binär) können keine dynamischen Variablen verarbeiten; in diesem Fall tritt ein Fehler auf. Große Variablen stellen kein Problem für diese Arbeitsdateitypen dar, außer wenn die maximale Feld-/Datensatzlänge von 32766 Bytes überschritten wird.

Besondere Bedingungen für TRANSFER und ENTIRE CONNECTION

In Verbindung mit dem Statement `READ WORK FILE` kann der Arbeitsdateityp TRANSFER dynamische Variablen verarbeiten. Es gibt keine Größenbeschränkung für dynamische Variablen. Der Arbeitsdateityp ENTIRE CONNECTION kann keine dynamischen Variablen verarbeiten. Beide Arbeitsdateitypen können jedoch große Variablen mit einer maximalen Feld-/Datensatzlänge von 1073741824 Bytes verarbeiten.

In Verbindung mit dem Statement `WRITE WORK FILE` kann der Arbeitsdateityp TRANSFER dynamische Variablen mit einer maximalen Feld-/Datensatzlänge von 32766 Bytes verarbeiten. Der Arbeitsdateityp ENTIRE CONNECTION kann keine dynamischen Variablen verarbeiten. Beide Arbeitsdateitypen können jedoch große Variablen mit einer maximalen Feld-/Datensatzlänge von 1073741824 Bytes verarbeiten.

DDM-Generierung und Editieren von Spalten mit variabler Länge

Abhängig von den Datentypen wird entweder das entsprechende Datenbankformat A oder B generiert. Für den Datenbank-Datentyp `VARCHAR` wird die Natural-Länge der Spalte auf die maximale Länge des Datentyps gesetzt, und zwar so wie sie im DBMS definiert ist. Wenn ein Datentyp sehr groß ist, wird an der Position des Längenfeldes das Schlüsselwort `DYNAMIC` generiert.

Für alle Spalten mit variabler Länge wird ein `LINDICATOR`-Feld `L@<column-name>` generiert. Für den Datenbank-Datentyp `VARCHAR` wird ein `LINDICATOR`-Feld mit Format/Länge I2 generiert. Bei großen Datentypen (siehe die Liste unten) wird Format/Länge I4 benutzt.

Bei einem Datenbankzugriff bietet `LINDICATOR` die Chance, die Länge des zu lesenden Feldes zu bekommen oder die Länge des zu schreibenden Feldes unabhängig von der definierten Pufferlänge

zu setzen (oder unabhängig von *LENGTH). Nach einer Abfragefunktion wird *LENGTH normalerweise auf den Wert des entsprechenden Längenindikators gesetzt.

Beispiel-DDM:

```
 T  L  Name                     F   Leng        S   D   Remark
    :
    1  L@PICTURE1               I   4                                     /*
length indicator
    1  PICTURE1                 B   DYNAMIC             IMAGE
    1  N@PICTURE1              I   2                                   /* NULL
indicator
    1  L@TEXT1                  I   4                                     /*
length indicator
    1  TEXT1                    A   DYNAMIC             TEXT
    1  N@TEXT1                 I   2                                   /* NULL
indicator
    1  L@DESCRIPTION            I   2                                     /*
length indicator
    1  DESCRIPTION              A   1000                VARCHAR(1000)
    :
    :
 ~~~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/*
concerning PICTURE1
   Header              :    ---
   Edit Mask           :    ---
   Remarks             :   IMAGE
```

Die generierten Formate sind von unterschiedlicher Länge. Ein Natural-Programmierer hat die Möglichkeit, die Definition von DYNAMIC in eine feste Längendefinition (erweiterte Feldbearbeitung) zu ändern und kann zum Beispiel die entsprechende DDM-Felddefinition für VARCHAR-Datentypen in ein Feld mit multiplen Werten abändern (alte Generierung).

Beispiel:

```
 T  L  Name                 F   Leng          S   D   Remark
    :
    1  L@PICTURE1           I   4                                         /*
length indicator
    1  PICTURE1             B   1000000000            IMAGE
    1  N@PICTURE1          I   2                                       /* NULL
indicator
    1  L@TEXT1              I   4                                         /*
length indicator
    1  TEXT1                A   5000                  TEXT
    1  N@TEXT1             I   2                                       /* NULL
indicator
    1  L@DESCRIPTION        I   2                                         /*
```

```
length indicator
  M  1  DESCRIPTION           A   100                   VARCHAR(1000)
       :
       :
  ~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/*
concerning PICTURE1
   Header             :    ---
   Edit Mask          :    ---
   Remarks            :   IMAGE
```

Zugriff auf große Datenbankobjekte

Für den Zugriff auf eine Datenbank mit großen Objekten (CLOBs oder BLOBs) ist ein DDM mit entsprechenden großen alphanumerischen Feldern, Unicode-Feldern oder binären Feldern erforderlich. Wenn eine feste Länge definiert ist und in der Datenbank ein großes Objekt nicht in dieses Feld passt, wird das große Objekt abgeschnitten. Wenn der Programmierer die eindeutige Länge des Datenbankobjektes nicht kennt, dann ist es sinnvoll mit dynamischen Feldern zu arbeiten. So viele Neuzuteilungen wie nötig werden durchgeführt, um das Objekt zu halten. Es wird nichts abgeschnitten.

Beispielprogramm:

```
DEFINE DATA LOCAL

1 person VIEW OF xyz-person
  2 last_name
  2 first_name_1
  2 L@PICTURE1                  /* I4 length indicator for PICTURE1
  2 PICTURE1                    /* defined as dynamic in the DDM
  2 TEXT1                       /* defined as non-dynamic in the DDM

END-DEFINE

SELECT * INTO VIEW person FROM xyz-person                   /* PICTURE1 will be
read completely
                             WHERE last_name = 'SMITH'      /* TEXT1 will be
truncated to fixed length 5000

   WRITE  'length of PICTURE1: '  L@PICTURE1              /* the L-INDICATOR will
contain the length
                                                            /* of PICTURE1 (=
*LENGTH(PICTURE1)
   /* do something with PICTURE1 and TEXT1

   L@PICTURE1  :=  100000
   INSERT INTO xyz-person  (*) VALUES (VIEW person)        /* only the first 100000
Bytes of PICTURE1
```

```
                                               /* are inserted
END-SELECT
```

Wenn im View eine Format-/Längendefinition weggelassen wird, wird sie aus dem DDM genommen. Im Reporting Mode ist es jetzt möglich, eine beliebige Länge zu definieren, wenn das entsprechende DDM-Feld als DYNAMIC definiert ist. Das dynamische Feld wird in einem Feld mit einer festen Pufferlänge abgebildet. Anders herum ist es nicht möglich.

DDM-Format-/Längendefinition	VIEW-Format-/Längendefinition	
(An)	-	gültig
	(An)	gültig
	(Am)	nur im Reporting Mode gültig
	(A) DYNAMIC	ungültig
(A) DYNAMIC	-	gültig
	(A) DYNAMIC	gültig
	(An)	nur im Reporting Mode gültig
	(Am / i : j)	nur im Reporting Mode gültig

(entsprechend für Format B-Variablen)

Parameter mit LINDICATOR-Klausel in SQL-Statements

Wenn das LINDICATOR-Feld als ein I2-Feld definiert ist, wird der SQL-Datentyp VARCHAR zum Senden oder Empfangen der entsprechenden Spalte benutzt. Wenn die LINDICATOR-Host-Variable als I4 angegeben ist, wird ein Datentyp für große Objekte (CLOB/BLOB) benutzt.

Wenn das Feld als DYNAMIC definiert ist, wird die Spalte bis zu ihrer tatsächlichen Länge in einer internen Schleife gelesen. Das LINDICATOR-Feld und *LENGTH werden auf diese Länge gesetzt. Bei einem Feld mit einer festen Länge wird die Spalte bis zur definierten Länge gelesen. In beiden Fällen wird das Feld bis zu dem im LINDICATOR-Feld definierten Wert geschrieben.

Performance-Aspekte bei dynamischen Variablen

Wenn eine dynamische Variable in kleinen Beträgen mehrmals (z.B. byteweise) erweitert werden soll, benutzen Sie das EXPAND-Statement vor den Schleifen-Iterationen, wenn die obere Grenze des erforderlichen Speichers (ungefähr) bekannt ist. Dadurch vermeiden Sie zusätzlichen Verarbeitungsmehraufwand zum Anpassen des erforderlichen Speicherplatzes.

Benutzen Sie das REDUCE- oder RESIZE-Statement, wenn die dynamische Variable nicht mehr erforderlich ist, insbesondere für Variablen mit einem hohen Wert von *LENGTH. Dadurch wird es Natural ermöglicht, den Speicherplatz wieder zu benutzen oder freizugeben. Somit kann die Gesamtleistung verbessert werden.

Die Größe des einer dynamischen Variable zugewiesenen Hauptspeichers kann mittels des `REDUCE DYNAMIC VARIABLE`-Statements auf eine angegebene Größe reduziert werden. Um eine Variable auf eine angegebene Größe (neu) zuzuweisen, kann das `EXPAND`-Statement benutzt werden.

Wenn die Variable initialisiert werden soll, benutzen Sie das `MOVE ALL UNTIL`-Statement.

Beispiel:

```
** Example 'DYNAMX06': Dynamic variables (allocated memory)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1  (A)  DYNAMIC
1 #LEN         (I4)
END-DEFINE
*
#MYDYNTEXT1 := 'a'     /* used length is 1, value is 'a'
                       /* allocated size is still 1
WRITE *LENGTH(#MYDYNTEXT1)
*
EXPAND DYNAMIC VARIABLE  #MYDYNTEXT1 TO 100
                       /* used length is still 1, value is 'a'
                       /* allocated size is 100
*
CALLNAT 'DYNAMX05' USING #MYDYNTEXT1
WRITE *LENGTH(#MYDYNTEXT1)
                       /* used length and allocated size
                       /* may have changed in the subprogram
*
#LEN := *LENGTH(#MYDYNTEXT1)
REDUCE DYNAMIC VARIABLE  #MYDYNTEXT1  TO  #LEN
                       /* if allocated size is greater than used length,
                       /* the unused memory is released
*
REDUCE DYNAMIC VARIABLE  #MYDYNTEXT1  TO  0
WRITE *LENGTH(#MYDYNTEXT1)
                       /* free allocated memory for dynamic variable
END
```

Regeln:

- Benutzen Sie dynamische Operanden überall dort, wo es sinnvoll ist.
- Benutzen Sie `EXPAND`, wenn die obere Grenze der Hauptspeicher-Benutzung bekannt ist.
- Benutzen Sie `REDUCE`, wenn der dynamische Operand nicht mehr gebraucht wird.

Ausgabe von dynamische Variablen

Dynamische Variablen können innerhalb von Ausgabe-Statements wie folgt benutzt werden:

Statement	Anmerkungen
DISPLAY	Mit diesen Statements müssen Sie das Format der Ausgabe oder Eingabe von dynamischen Variablen setzen, indem Sie die Session-Parameter AL (Alphanumerische Länge für die Ausgabe) oder EM (Editiermaske) benutzen.
WRITE	
INPUT	
REINPUT	--
PRINT	Da die Ausgabe des PRINT-Statements unformatiert ist, muss die Ausgabe der dynamischen Variablen im PRINT-Statement nicht mittels der Parameter AL und EM gesetzt werden. Mit anderen Worten: diese Parameter können weggelassen werden.

Dynamische X-Arrays

Ein dynamisches X-Array kann zugewiesen werden, indem man zuerst die Anzahl der Ausprägungen angibt und dann die Länge der vorher zugewiesenen Array-Ausprägungen erweitert.

Beispiel:

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
*
EXPAND  ARRAY   #X-ARRAY TO (1:10)  /* Current boundaries (1:10)
#X-ARRAY(*) := 'ABC'
EXPAND  ARRAY   #X-ARRAY TO (1:20)  /* Current boundaries (1:20)
#X-ARRAY(11:20) := 'DEF'
```

21 Benutzerkonstanten

- Numerische Konstanten 142
- Alphanumerische Konstanten 143
- Unicode-Konstanten 145
- Datums- und Zeitkonstanten 148
- Hexadezimale Konstanten 150
- Logische Konstanten 152
- Gleitkomma-Konstanten 152
- Attribut-Konstanten 153
- Handle-Konstanten 154
- Namens-Konstanten definieren 154

Konstanten können überall in Natural-Programmen benutzt werden. Dieses Dokument behandelt die Arten von Konstanten, die unterstützt werden, und erläutert wie sie benutzt werden.

Numerische Konstanten

Folgende Themen werden behandelt:

- Numerische Konstanten
- Verarbeitung von numerischen Konstanten

Numerische Konstanten

Eine numerische Konstante kann 1 bis 29 numerische Ziffern enthalten.

Eine mit einem `COMPUTE`-, `MOVE`- oder arithmetischen Statement benutzte numerische Konstante kann einen Dezimalpunkt und Vorzeichen enthalten.

Beispiele:

```
1234   +1234   -1234

12.34  +12.34  -12.34
```

```
MOVE 3 TO #XYZ
COMPUTE #PRICE = 23.34
COMPUTE #XYZ = -103
COMPUTE #A = #B * 6074
```

Anmerkung: Intern werden numerische Konstanten ohne Dezimalziffern in Ganzzahl-Form (Format I) dargestellt, während numerische Konstanten mit Dezimalziffern sowie numerische Konstanten ohne Dezimalziffern, die zu groß sind, um in Format I zu passen, in gepackter Form (Format P) dargestellt werden.

Beispiel:

Numerische Konstante		Format	Länge
von	bis		
	<= -2147483649	P	>=10
-2147483648	-32769	I	4
-32768	32767	I	2
32768	2147483647	I	4
>= 2147483648		P	>=10

Verarbeitung von numerischen Konstanten

Wenn eine numerische Konstante innerhalb eines der Statements `COMPUTE`, `MOVE` oder `DEFINE DATA` mit der `INIT`-Option benutzt werden, überprüft Natural zur Kompilierungszeit, ob ein Konstanten-Wert in das entsprechende Feld passt. Dadurch werden Laufzeitfehler in Situationen vermieden, in denen eine solche Fehlerbedingung bereits bei der Kompilierung entdeckt werden kann.

Alphanumerische Konstanten

Folgende Themen werden behandelt:

- Struktur einer alphanumerischen Konstanten
- Verwendung von Apostrophen in alphanumerischen Konstanten
- Verketten von alphanumerischen Konstanten

Struktur einer alphanumerischen Konstanten

Eine alphanumerische Konstante kann 1 bis 1.073.741.824 Bytes (1 GB) alphanumerische Zeichen enthalten. Eine alphanumerische

Eine alphanumerische Konstante muss entweder in Apostrophen (')

```
'text'
```

oder Anführungszeichen (") stehen.

```
"text"
```

Beispiele:

```
MOVE 'ABC' TO #FIELDX
MOVE '% INCREASE' TO #TITLE
DISPLAY "LAST-NAME" NAME
```

Anmerkung: Eine alphanumerische Konstante, die zur Zuweisung eines Wertes zu einer **Benutzervariable** verwendet wird, darf nicht auf mehrere Statement-Zeilen aufgeteilt werden.

Verwendung von Apostrophen in alphanumerischen Konstanten

Möchten Sie, dass ein Apostroph (') Teil einer in Apostrophen stehenden alphanumerischen Konstante wird, müssen Sie dies durch 2 Apostrophen (' ') oder ein einzelnes Anführungszeichen (") ausdrücken.

Möchten Sie, dass ein Apostroph (') Teil einer in Anführungszeichen (") stehenden alphanumerischen Konstante wird, drücken Sie dies durch einen einzelnen Apostroph (') aus.

Beispiel:

Wenn Sie Folgendes ausgeben möchten

```
HE SAID, 'HELLO'
```

können Sie eine der folgenden Notationen benutzen:

```
WRITE 'HE SAID, ''HELLO'''
WRITE 'HE SAID, "HELLO"'
WRITE "HE SAID, ""HELLO"""
WRITE "HE SAID, 'HELLO'"
```

Anmerkung: Wenn Anführungszeichen nicht in Apostrophen konvertiert werden (siehe oben), dann ist der Grund dafür die Einstellung des Profilparameters `TQ` (Anführungszeichen konvertieren). Einzelheiten dazu erfahren Sie von Ihrem Natural-Administrator.

Verketten von alphanumerischen Konstanten

Alphanumerische Konstanten können mittels eines Bindestrichs miteinander verkettet werden, so dass sie einen einzelnen Wert bilden.

Beispiele:

```
MOVE 'XXXXXX' - 'YYYYYY' TO #FIELD
MOVE "ABC" - 'DEF' TO #FIELD
```

Auf diese Art können alphanumerische Konstanten auch mit **hexadezimalen Konstanten** verkettet werden.

Unicode-Konstanten

Folgende Themen werden behandelt:

- Unicode-Textkonstanten
- Apostroph innerhalb von Unicode-Textkonstanten
- Unicode-Hexadezimalkonstanten
- Verketten von Unicode-Konstanten

Unicode-Textkonstanten

Einer Unicode-Textkonstante muss das Zeichen `U` vorausgehen, und sie muss entweder in Apostrophen (') stehen

```
U'text'
```

oder in Anführungszeichen ("):

```
U"text"
```

Beispiel:

```
U'HELLO'
```

Der Compiler speichert diese Textkonstante in dem generierten Programm im Unicode-Format (UTF-16).

Apostroph innerhalb von Unicode-Textkonstanten

Wenn Sie möchten, dass ein Apostroph (') Teil einer Unicode-Textkonstante wird, die in Apostrophen steht, müssen Sie dies als zwei Apostrophe (' ') oder als ein einzelnes Anführungszeichen (") kodieren.

Wenn Sie möchten, dass ein Apostroph Teil einer Unicode-Textkonstante wird, die in Anführungszeichen steht, müssen Sie dies als einen einzelnen Apostroph (') kodieren.

Beispiel:

Wenn Sie Folgendes ausgeben möchten

```
HE SAID, 'HELLO'
```

können Sie eine der folgenden Notationen benutzen:

```
WRITE U'HE SAID, ''HELLO'''
WRITE U'HE SAID, "HELLO"'
WRITE U"HE SAID, ""HELLO"""
WRITE U"HE SAID, 'HELLO'"
```

Anmerkung: Wenn Anführungszeichen nicht in Apostrophe umgesetzt werden (siehe oben), dann liegt dies an der Einstellung des Profilparameters `TQ` (Anführungszeichen umsetzen). Einzelheiten erfahren Sie von Ihrem Natural-Administrator.

Unicode-Hexadezimalkonstanten

Die folgende Syntax wird benutzt, um ein Unicode-Zeichen oder eine Unicode-Zeichenkette in seiner hexadezimale Notation anzugeben:

```
UH'hhhh...'
```

wobei `h` eine hexadezimale Ziffer (0−9, A−F) ist.

Da ein UTF-16-Unicode-Zeichen aus einem Doppelbyte besteht, muss die Anzahl der angegebenen hexadezimalen Zeichen ein Mehrfaches von vier sein.

Beispiel:

Dieses Beispiel definiert die Zeichenkette `45`.

```
UH'00340035'
```

Verketten von Unicode-Konstanten

Die Verkettung von Unicode-Textkonstanten (U) und Unicode-Hexadezimalkonstanten (UH) ist zulässig.

Gültiges Beispiel:

```
MOVE U'XXXXXX' - UH'00340035' TO #FIELD
```

Unicode-Textkonstanten oder Unicode-Hexadezimalkonstanten können nicht mit alphanumerischen Codepage-Konstanten oder H-Konstanten verkettet werden.

Ungültiges Beispiel:

```
MOVE U'ABC' - 'DEF' TO #FIELD
MOVE UH'00340035' - H'414243' TO #FIELD
```

Weiteres gültiges Beispiel:

```
DEFINE DATA LOCAL
1 #U10 (U10)              /* Unicode variable with 10 (UTF-16) characters,
                          /* total byte length = 20
1 #UD  (U)  DYNAMIC       /* Unicode variable with dynamic length
END-DEFINE
*
#U10 := U'ABC'           /* Constant is created as X'004100420043' in the object,
                          /* the UTF-16 representation for string 'ABC'.

#U10 := UH'004100420043'  /* Constant supplied in hexadecimal format only,
                          /* corresponds to U'ABC'

#U10 := U'A'-UH'0042'-U'C' /* Constant supplied in mixed formats, corresponds to
U'ABC'.
END
```

Datums- und Zeitkonstanten

Folgende Themen werden behandelt:

- Datumskonstante
- Zeitkonstante
- Erweiterte Zeitkonstante

Datumskonstante

Eine Datumskonstante kann in Verbindung mit einer Variablen des Formats D benutzt werden.

Datumskonstanten können folgende Formate haben:

`D'`*yyyy-mm-dd*`'`	Internationales Datumsformat
`D'`*dd.mm.yyyy*`'`	Deutsches Datumsformat
`D'`*dd/mm/yyyy*`'`	Europäisches Datumsformat
`D'`*mm/dd/yyyy*`'`	USA-Datumsformat

Dabei bezeichnet *dd* den Tag, *mm* den Monat und *yyyy* das Jahr.

Beispiel:

```
DEFINE DATA LOCAL
  1 #DATE (D)
END-DEFINE
...
MOVE D'2004-03-08' TO #DATE
...
```

Das standardmäßige Datumsformat wird von dem vom Natural-Administrator gesetzten Profilparameter `DTFORM` (Datumsformat) gesteuert.

Zeitkonstante

Eine Zeitkonstante kann in Verbindung mit einer Variablen des Formats T benutzt werden.

Eine Zeitkonstante hat das folgende Format:

```
T'hh:ii:ss'
```

Dabei bezeichnet *hh* Stunden, *ii* Minuten und *ss* Sekunden.

Beispiel:

```
DEFINE DATA LOCAL
  1 #TIME (T)
END-DEFINE
...
MOVE T'11:33:00' TO #TIME
...
```

Erweiterte Zeitkonstante

Eine Zeitvariable (Format T) kann Datums- und Zeit-Informationen enthalten, wobei die Datumsinformationen eine Untermenge der Zeitinformationen sind. Allerdings können bei einer „normalen“ Zeitkonstante (Präfix `T`) nur die Zeit-Informationen einer Zeitvariablen verarbeitet werden:

```
T'hh:ii:ss'
```

Bei einer erweiterten Zeitkonstante (Präfix E) ist es möglich, den vollständigen Inhalt einer Zeitvariablen einschließlich der Datums-Informationen zu verarbeiten:

```
E'yyyy-mm-dd hh:ii:ss'
```

Einmal abgesehen davon ist die Benutzung einer erweiterten Zeitkonstanten in Verbindung mit einer Zeitvariablen identisch mit der für eine normale Zeitkonstante.

Anmerkung: Das Format, in dem Datums-Informationen angegeben werden müssen, ist bei einer erweiterten Zeitkonstante abhängig von der Einstellung des Profilparameters DTFORM. Bei der oben angegebenen erweiterten Zeitkonstante wird von DTFORM=I (internationales Datumsformat) ausgegangen.

Hexadezimale Konstanten

Folgende Themen werden behandelt:

- Verwundung und Verarbeitung von hexadezimalen Konstanten
- Verketten von hexadezimalen Konstanten

Verwundung und Verarbeitung von hexadezimalen Konstanten

Eine hexadezimale Konstante kann benutzt werden, um einen Wert einzugeben, der nicht als ein standardmäßiges Tastaturzeichen eingegeben werden kann.

Eine hexadezimale Konstante kann 1 bis 1.073.741.824 bytes (1 GB) alphanumerische Zeichen enthalten.

Einer hexadezimalen Konstante geht ein Präfix H voraus. Die Konstante selbst muss in Apostrophen (') stehen und kann aus den hexadezimalen Zeichen 0 – 9, A – F bestehen. Zwei hexadezimale Zeichen sind erforderlich, um ein Daten-Byte darzustellen.

Die hexadezimale Darstellung eines Zeichens variiert, je nachdem, ob Ihr Computer einen ASCII- oder EBCDIC-Zeichensatz verwendet. Wenn Sie hexadezimale Konstanten auf einen anderen Computer übertragen, müssen Sie deshalb vielleicht die Zeichen konvertieren.

ASCII-Beispiele:

```
H'313233'    (equivalent to the alphanumeric constant '123')
H'414243'    (equivalent to the alphanumeric constant 'ABC')
```

EBCDIC-Beispiele:

```
H'F1F2F3'    (equivalent to the alphanumeric constant '123')
H'C1C2C3'    (equivalent to the alphanumeric constant 'ABC')
```

Wenn eine hexadezimale Konstante in ein anderes Feld übertragen wird, wird sie als ein alphanumerischer Wert behandelt (Format A).

Die Datenübertragung eines alphanumerischen Werts (Format A) in ein Feld, das nicht in einem der Formate A, U oder B definiert ist, ist nicht zulässig. Deshalb wird eine hexadezimale Konstante als Ausgangswert in einem `DEFINE DATA`-Statement mit einem Syntaxfehler NAT0094 zurückgewiesen, wenn die entsprechende Variable nicht vom Format A, U oder B ist.

Beispiel:

```
DEFINE DATA LOCAL
1 #I(I2) INIT <H'000F'>      /* causes a NAT0094 syntax error
END-DEFINE
```

Verketten von hexadezimalen Konstanten

Hexadezimale Konstanten können mittels eines Bindestrichs zwischen den Konstanten miteinander verkettet werden.

ASCII-Beispiel:

```
H'414243' - H'444546' (equivalent to 'ABCDEF')
```

EBCDIC-Beispiel:

```
H'C1C2C3' - H'C4C5C6' (equivalent to 'ABCDEF')
```

Auf diese Weise können hexadezimale Konstanten auch mit alphanumerischen Konstanten verkettet werden.

Logische Konstanten

Die logischen Konstanten TRUE (wahr) und FALSE (falsch) können benutzt werden, um einen logischen Wert einem mit Format L definierten Feld zuzuweisen.

Beispiel:

```
DEFINE DATA LOCAL
  1 #FLAG (L)
END-DEFINE
...
MOVE TRUE TO #FLAG
...
IF #FLAG ...
  statement ...
  MOVE FALSE TO #FLAG
END-IF
 ...
```

Gleitkomma-Konstanten

Gleitkomma-Konstanten können mit im Format F definierten Variablen benutzt werden.

Beispiel:

```
DEFINE DATA LOCAL
  1 #FLT1 (F4)
END-DEFINE
...
COMPUTE #FLT1 = -5.34E+2
...
```

Attribut-Konstanten

Attribut-Konstanten können mit im Format C (Kontroll-Variablen) definierten Variablen benutzt werden. Diese Art von Konstante muss in Klammern stehen.

Die folgenden Attribute können benutzt werden:

Attribut	Beschreibung
AD=D	Standard
AD=B	blinkend
AD=I	hell hervorgehoben
AD=N	keine Anzeige
AD=V	invers
AD=U	unterstrichen
AD=C	kursiv
AD=Y	dynamisches Attribut
AD=P	geschützt
CD=BL	blau
CD=GR	grün
CD=NE	neutral
CD=PI	rosa
CD=RE	rot
CD=TU	türkis
CD=YE	gelb

Weitere Informationen zu diesen Attributen finden Sie bei den Session-Parametern AD und CD.

Beispiel:

```
DEFINE DATA LOCAL
  1 #ATTR (C)
  1 #FIELD (A10)
END-DEFINE
...
MOVE (AD=I CD=BL) TO #ATTR
...
INPUT #FIELD (CV=#ATTR)
...
```

Handle-Konstanten

Die Handle-Konstante `NULL-HANDLE` kann mit GUI-Handles und Objekt-Handles benutzt werden.

Weitere Informationen zu GUI-Handles siehe ***How To Define Dialog Elements***.

Weitere Informationen zu Objekt-Handles siehe ***NaturalX***.

Namens-Konstanten definieren

Wenn Sie denselben Konstanten-Wert mehrmals in einem Programm benutzen müssen, können Sie den Pflegeaufwand durch Definition einer Namens-Konstante reduzieren:

- Definieren Sie ein Feld im `DEFINE DATA`-Statement,
- weisen Sie ihm einen Konstanten-Wert zu und
- benutzen Sie im Programm den Feldnamen anstatt des Konstanten-Werts.

Wenn der Wert geändert werden muss, müssen Sie ihn somit nur einmal im `DEFINE DATA`-Statement und nicht überall in dem Programm ändern, in dem er vorkommt.

Geben Sie den Konstanten-Wert in viereckigen Klammern mit dem **Schlüsselwort** `CONSTANT` hinter der Feld-Definition im `DEFINE DATA`-Statement an.

- Wenn der Wert alphanumerisch ist, muss er in Apostrophen (') stehen.
- Wenn der Wert Text im Unicode-Format ist, muss ihm das Zeichen `U` vorangehen, und er muss in Apostrophen (') stehen.
- Wenn der Wert im hexadezimalen Unicode-Format ist, müssen ihm die Zeichen `UH` vorangehen, und er muss in Apostrophen (') stehen.

Beispiel:

```
DEFINE DATA LOCAL
  1 #FIELDA (N3) CONSTANT <100>
  1 #FIELDB (A5) CONSTANT <'ABCDE'>
  1 #FIELDC (U5) CONSTANT <U'ABCDE'>
  1 #FIELDD (U5) CONSTANT <UH'00410042004300440045'>
END-DEFINE
...
```

Während der Ausführung des Programms kann der Wert einer solchen Namens-Konstante nicht geändert werden.

22 Ausgangswerte (und das RESET-Statement)

- Standard-Ausgangswert einer Benutzervariablen/ eines Arrays 156
- Ausgangswert einer Benutzervariablen/einem Array zuweisen 156
- Benutzervariable auf ihren Ausgangswert zurücksetzen 159

Dieses Kapitel beschreibt die standardmäßigen Ausgangswerte von Benutzervariablen, erläutert, wie Sie einer Benutzervariable einen Ausgangswert zuweisen können und wie Sie das RESET-Statement zum Zurücksetzen des Feldwertes auf seinen Standard-Ausgangswert oder den für diese Variable im DEFINE DATA-Statement definierten Ausgangswert benutzen können.

Standard-Ausgangswert einer Benutzervariablen/ eines Arrays

Wenn Sie für ein Feld keinen Ausgangswert angeben, wird das Feld je nach seinem Format mit einem Standard-Ausgangswert initialisiert:

Format	Standard-Ausgangswert
B, F, I, N, P	0
A, U	Leerzeichen
L	F(ALSE)
D	D' '
T	T'00:00:00'
C	(AD=D)
GUI Handle	NULL-HANDLE
Object Handle	NULL-HANDLE

Ausgangswert einer Benutzervariablen/einem Array zuweisen

Im DEFINE DATA-Statement können Sie einer Benutzervariable einen Ausgangswert zuweisen. Wenn der Ausgangswert alphanumerisch ist, muss er in Apostrophen (') stehen.

Folgende Themen werden behandelt:

- Änderbaren Ausgangswert zuweisen
- Konstanten-Ausgangswert zuweisen
- Natural-Systemvariable als Ausgangswert zuweisen

- Zeichen als Ausgangswert für alphanumerische Variable zuweisen

Änderbaren Ausgangswert zuweisen

Wenn der Variablen bzw. dem Array ein änderbarer Ausgangswert zugewiesen werden soll, geben Sie den Ausgangswert in spitzen Klammern mit dem Schlüsselwort INIT nach der Variablen-Definition im DEFINE DATA-Statement an. Die zugewiesenen Werte werden jedesmal benutzt, wenn die Variable bzw. das Array referenziert wird. Die zugewiesenen Werte können während der Programmausführung geändert werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3) INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
END-DEFINE
...
```

Konstanten-Ausgangswert zuweisen

Wenn die Variable bzw. das Array als eine Namens-Konstante behandelt werden soll, geben Sie den Ausgangswert in spitzen Klammern mit dem Schlüsselwort CONSTANT nach der Variablen-Definition im DEFINE DATA-Statement an. Die zugewiesenen Konstanten-Werte werden jedesmal benutzt, wenn die die Variable bzw. das Array referenziert wird. Die zugewiesenen Werte können während der Programmausführung *nicht* geändert werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3) CONST <100>
1 #FIELDB (A20) CONST <'ABC'>
END-DEFINE
...
```

Natural-Systemvariable als Ausgangswert zuweisen

Als Ausgangswert für ein Feld kann auch der Wert einer **Natural-Systemvariablen** genommen werden.

Beispiel:

Hier liefert die Systemvariable `*DATX` den Ausgangswert.

```
DEFINE DATA LOCAL
1 #MYDATE (D) INIT <*DATX>
END-DEFINE
...
```

Zeichen als Ausgangswert für alphanumerische Variable zuweisen

Als Ausgangswert können Sie auch eine Variable vollständig oder teilweise mit einem bestimmten Zeichen oder einer Zeichenkette füllen (nur bei alphanumerischen Variablen möglich).

- **Feld komplett füllen:**
  Mit der Option `FULL LENGTH <character(s)>` wird das gesamte Feld mit dem/den angegebenen Zeichen gefüllt.

  Hier wird das gesamte Feld mit Sternen (*) gefüllt:

  ```
  DEFINE DATA LOCAL
  1 #FIELD (A25) INIT FULL LENGTH <'*'>
  END-DEFINE
  ...
  ```

- **Ersten *n* Stellen eines Feldes füllen:**
  Mit der Option `LENGTH n <character(s)>` werden die ersten *n* Stellen des Feldes mit dem/den angegebenen Zeichen gefüllt.

  Hier werden die ersten 4 Stellen des Feldes mit Ausrufungszeichen gefüllt.

  ```
  DEFINE DATA LOCAL
  1 #FIELD (A25) INIT LENGTH 4 <'!'>
  END-DEFINE
  ...
  ```

Benutzervariable auf ihren Ausgangswert zurücksetzen

Das RESET-Statement dient dazu, den Wert eines Feldes zurückzusetzen. Zwei Optionen stehen dabei zur Verfügung:

- Auf Standard-Ausgangswert zurücksetzen
- Auf den im DEFINE DATA-Statement definierten Ausgangswert zurücksetzen

Anmerkungen:

1. Ein mit einer CONSTANT-Klausel im DEFINE DATA-Statement deklariertes Feld kann nicht in einem RESET-Statement referenziert werden, weil sein Inhalt nicht geändert werden kann.
2. Im Reporting Mode dient das RESET-Statement auch zur Definition einer Variablen, vorausgesetzt das Programm enthält kein DEFINE DATA LOCAL-Statement.

Auf Standard-Ausgangswert zurücksetzen

RESET (ohne INITIAL) setzt den Inhalt jedes angegebenen Feldes je nach Format auf seinen Standard-Ausgangswert zurück.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3)  INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
1 #FIELDC (I4)  INIT <5>
END-DEFINE
...
...
RESET #FIELDA                    /* resets field value to default initial value
...
```

Auf den im DEFINE DATA-Statement definierten Ausgangswert zurücksetzen

RESET INITIAL setzt den Inhalt jedes angegebenen Feldes auf den Ausgangswert zurück, der für das Feld im DEFINE DATA-Statement definiert wurde.

Für ein ohne INIT-Klausel im DEFINE DATA-Statement deklariertes Feld hat RESET INITIAL denselben Effekt wie RESET (ohne INITIAL).

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3)  INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
1 #FIELDC (I4)  INIT <5>
END-DEFINE
...
RESET INITIAL #FIELDA #FIELDB #FIELDC /* resets field values to initial values as
defined in DEFINE DATA
...
```

23 Felder redefinieren

- REDEFINE-Option des DEFINE DATA-Statements 162
- Beispielprogramm für eine Redefinition 164

Die Redefinition dient dazu, das Format eines Feldes zu ändern oder ein einzelnes Feld in mehrere Teile aufzuteilen.

REDEFINE-Option des DEFINE DATA-Statements

Mit der REDEFINE-Option des DEFINE DATA-Statements kann ein einzelnes Feld — entweder eine Benutzervariable oder ein Datenbankfeld — als ein neues Feld oder mehrere neue Felder redefiniert werden. Eine Gruppe kann ebenfalls redefiniert werden.

Wichtig: Dynamische Variablen sind bei einer Redefinition nicht zulässig.

Die REDEFINE-Option redefiniert die Byte-Positionen eines Feldes von links nach rechts, unabhängig vom Format. Die Byte-Positionen des ursprünglichen Feldes und des neudefinierten Feldes bzw. der neudefinierten Felder müssen einander entsprechen.

Eine Redefinition muss unmittelbar nach der Definition des ursprünglichen Feldes angegeben werden.

Beispiel 1:

Im folgenden Beispiel wird das Datenbankfeld BIRTH als drei neue Benutzervariablen redefiniert:

```
DEFINE DATA LOCAL
01 EMPLOY-VIEW VIEW OF STAFFDDM
  02 NAME
  02 BIRTH
  02 REDEFINE BIRTH
     03 #BIRTH-YEAR  (N4)
     03 #BIRTH-MONTH (N2)
     03 #BIRTH-DAY   (N2)
END-DEFINE
   ...
```

Beispiel 2:

Im folgenden Beispiel wird die Gruppe #VAR2, die aus zwei Benutzervariablen mit Format N bzw. P besteht, als eine neue Variable vom Format A redefiniert:

```
DEFINE DATA LOCAL
01 #VAR1 (A15)
01 #VAR2
  02 #VAR2A (N4.1)
  02 #VAR2B (P6.2)
01 REDEFINE #VAR2
  02 #VAR2RD (A10)
END-DEFINE
   ...
```

Mit der Notation FILLER *n*X können Sie in dem Feld, das redefiniert wird, *n* Füllbytes - d.h. Segmente, die nicht benutzt werden sollen - definieren. (Nachgestellte Füllbytes müssen nicht unbedingt angegeben werden.)

Beispiel 3:

Im folgenden Beispiel wird die Benutzervariable #FIELD als drei neue Benutzervariablen, jede mit Format/Länge A2, redefiniert. Die FILLER-Notationen bedeuten, dass das 3. und 4. sowie das 7. bis 10. Byte des ursprünglichen Feldes nicht benutzt werden sollen.

```
DEFINE DATA LOCAL
1 #FIELD (A12)
1 REDEFINE #FIELD
  2 #RFIELD1 (A2)
  2 FILLER 2X
  2 #RFIELD2 (A2)
  2 FILLER 4X
  2 #RFIELD3 (A2)
END-DEFINE
...
```

Beispielprogramm für eine Redefinition

Das folgende Programm veranschaulicht die Anwendung einer Redefinition:

```
** Example 'DDATAX01': DEFINE DATA
************************************************************************
DEFINE DATA LOCAL
01 VIEWEMP VIEW OF EMPLOYEES
  02 NAME
  02 FIRST-NAME
  02 SALARY   (1:1)
*
01 #PAY       (N9)
01 REDEFINE #PAY
  02 FILLER   3X
  02 #USD     (N3)
  02 #OOO     (N3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  MOVE SALARY (1) TO #PAY
  DISPLAY NAME FIRST-NAME #PAY #USD #OOO
END-READ
END
```

Ausgabe des Programms `DDATAX01`:

Beachten Sie, wie das Feld `#PAY` und die aus seiner Redefinition resultierenden Felder angezeigt werden:

```
Page      1                                               04-11-11  14:15:54

        NAME              FIRST-NAME         #PAY    #USD #OOO
-------------------- -------------------- ---------- ---- ----

JONES                VIRGINIA                  46000   46    0
JONES                MARSHA                    50000   50    0
JONES                ROBERT                    31000   31    0
```

24 Arrays

- Arrays definieren 166
- Ausgangswerte für Arrays 167
- Ausgangswerte für eindimensionale Arrays zuweisen 167
- Ausgangswerte für zweidimensionale Arrays zuweisen 168
- Dreidimensionales Array 173
- Arrays als Teil einer größeren Datenstruktur 174
- Datenbank-Arrays 175
- Arithmetische Ausdrücke in Index-Notationen 175
- Arithmetische Funktionen bei Arrays 176

Natural unterstützt die Verarbeitung von sogenannten Arrays. Arrays sind mehrdimensionale Tabellen, d.h. zwei oder mehr logisch verwandte Elemente, die unter einem gemeinsamen Namen definiert werden.

Arrays können aus einzelnen Datenelementen mit mehreren Dimensionen bestehen oder aus hierarchischen Datenstrukturen, die sich wiederholende Strukturen oder individuelle Elemente aufweisen.

Arrays definieren

Ein Natural-Array kann ein-, zwei- oder dreidimensional sein. Es kann eine unabhängige Variable, Teil einer größeren Datenstruktur oder Teil einer Datenbanksicht sein.

Wichtig: Dynamische Variablen sind in einer Array-Definition nicht zulässig.

Um ein eindimensionales Array zu definieren

- Geben Sie hinter Format und Länge einen Schrägstrich (/) und danach eine Index-Notation, d.h. die Anzahl der Ausprägungen des Arrays, an.

  Das folgende Array hat zum Beispiel drei Ausprägungen, wobei jede Ausprägung Format/Länge A10 hat:

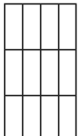
```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3)
END-DEFINE
...
```

Um ein zweidimensionales Array zu definieren

- Geben Sie für beide Dimensionen eine Index-Notation an:

```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3,1:4)
END-DEFINE
...
```

Ein zweidimensionales Array kann man sich als Tabelle vorstellen. Das im obigen Beispiel definierte Array wäre demnach eine Tabelle, die aus 3 Zeilen und 4 Spalten besteht:



Ausgangswerte für Arrays

Um einer oder mehreren Ausprägungen eines Arrays einen Ausgangswert zuzuweisen, verwenden Sie, ähnlich wie für „einfache“ Variablen (siehe folgende Beispiele), eine `INIT`-Angabe.

Ausgangswerte für eindimensionale Arrays zuweisen

Die folgenden Beispiele veranschaulichen, wie einem eindimensionalen Array Ausgangswerte zugewiesen werden:

- Um einer einzelnen Ausprägung einen Ausgangswert zuzuweisen, geben Sie an:

  ```
  1 #ARRAY (A1/1:3) INIT (2) <'A'>
  ```

  `A` wird der zweiten Ausprägung zugewiesen.
- Um allen Ausprägungen den gleichen Ausgangswert zuzuweisen, geben Sie an:

  ```
  1 #ARRAY (A1/1:3) INIT ALL <'A'>
  ```

  `A` wird jeder Ausprägung zugewiesen. Stattdessen könnten Sie auch angeben:

  ```
  1 #ARRAY (A1/1:3) INIT (*) <'A'>
  ```

- Um einem Bereich von mehreren Ausprägungen den gleichen Ausgangswert zuzuweisen, geben Sie an:

  ```
  1 #ARRAY (A1/1:3) INIT (2:3) <'A'>
  ```

  A wird der zweiten bis dritten Ausprägung zugewiesen.

- Um jeder Ausprägung einen anderen Ausgangswert zuzuweisen, geben Sie an:

  ```
  1 #ARRAY (A1/1:3) INIT <'A','B','C'>
  ```

  A wird der ersten Ausprägung zugewiesen, B der zweiten und C der dritten.

- Um verschiedenen (aber nicht allen) Ausprägungen verschiedene Ausgangswerte zuzuweisen, geben Sie an:

  ```
  1 #ARRAY (A1/1:3) INIT (1) <'A'> (3) <'C'>
  ```

  A wird der ersten Ausprägung zugewiesen und C der dritten; der zweiten Ausprägung wird kein Wert zugewiesen.

  Stattdessen könnten Sie auch angeben:

  ```
  1 #ARRAY (A1/1:3) INIT <'A',,'C'>
  ```

- Wenn weniger Ausgangswerte angegeben werden als Ausprägungen vorhanden sind, bleiben die letzten Ausprägungen leer:

  ```
  1 #ARRAY (A1/1:3) INIT <'A','B'>
  ```

  A wird der ersten Ausprägung zugewiesen und B der zweiten; der dritten Ausprägung wird kein Wert zugewiesen

Ausgangswerte für zweidimensionale Arrays zuweisen

Dieser Abschnitt zeigt, wie einem zweidimensionalen Array Ausgangswerte zugewiesen werden. Die folgenden Themen werden behandelt:

- Vorbemerkung
- Gleichen Wert zuweisen

- Unterschiedliche Werte zuweisen

Vorbemerkung

Für die Beispiele gehen wir von einem zweidimensionalen Array aus, das drei Ausprägungen in der ersten Dimension (Zeilen) und vier Ausprägungen in der zweiten Dimension (Spalten) hat:

```
1 #ARRAY (A1/1:3,1:4)
```

Vertikal: Erste Dimension (1:3), Horizontal: Zweite Dimension (1:4):

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

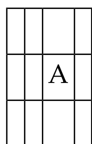
Die erste Gruppe von Beispielen veranschaulicht, wie den Ausprägungen eines zweidimensionalen Arrays der *gleiche* Ausgangswert zugewiesen wird; die zweite Gruppe von Beispielen veranschaulicht, wie *unterschiedliche* Ausgangswerte zugewiesen werden.

Beachten Sie bei den Beispielen insbesondere die Verwendung der Notationen `*` und `V`. Beide Notationen beziehen sich auf alle Ausprägungen der betreffenden Dimension: Mit `*` werden *alle* Ausprägungen der betreffenden Dimension mit dem gleichen Wert initialisiert, mit `V` werden alle Ausprägungen der betreffenden Dimension mit *unterschiedlichen* Werten initialisiert.

Gleichen Wert zuweisen

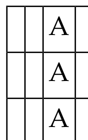
- Um einer Ausprägung einen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (2,3) <'A'>
```



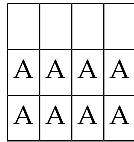
- Um einer Ausprägung der zweiten Dimension — in allen Ausprägungen der ersten Dimension — den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,3) <'A'>
```



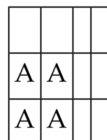
- Um einem Bereich von Ausprägungen der ersten Dimension — in allen Ausprägungen der zweiten Dimension — den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,*) <'A'>
```



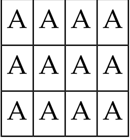
- Um einem Bereich von Ausprägungen in beiden Dimensionen den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,1:2) <'A'>
```



- Um allen Ausprägungen (in beiden Dimensionen) den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT ALL <'A'>
```



Stattdessen könnten Sie auch angeben:

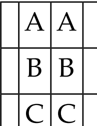
```
1 #ARRAY (A1/1:3,1:4) INIT (*,*) <'A'>
```

Unterschiedliche Werte zuweisen

- ```
  1 #ARRAY (A1/1:3,1:4) INIT (V,2) <'A','B','C'>
  ```

	A		
	B		
	C		

- ```
  1 #ARRAY (A1/1:3,1:4) INIT (V,2:3) <'A','B','C'>
  ```



- ```
  1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B','C'>
  ```

A	A	A	A
B	B	B	B
C	C	C	C

- ```
  1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A',,'C'>
  ```

A	A	A	A
C	C	C	C

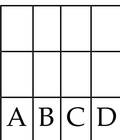
- ```
  1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B'>
  ```

A	A	A	A
B	B	B	B
  ```

```
1 #ARRAY (A1/1:3,1:4) INIT (V,1) <'A','B','C'> (V,3) <'D','E','F'>
```

A		D	
B		E	
C		F	

```
1 #ARRAY (A1/1:3,1:4) INIT (3,V) <'A','B','C','D'>
```



```
1 #ARRAY (A1/1:3,1:4) INIT (*,V) <'A','B','C','D'>
```

A	B	C	D
A	B	C	D
A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (*,2) <'B'> (3,3) <'C'> (3,4) <'D'>
```

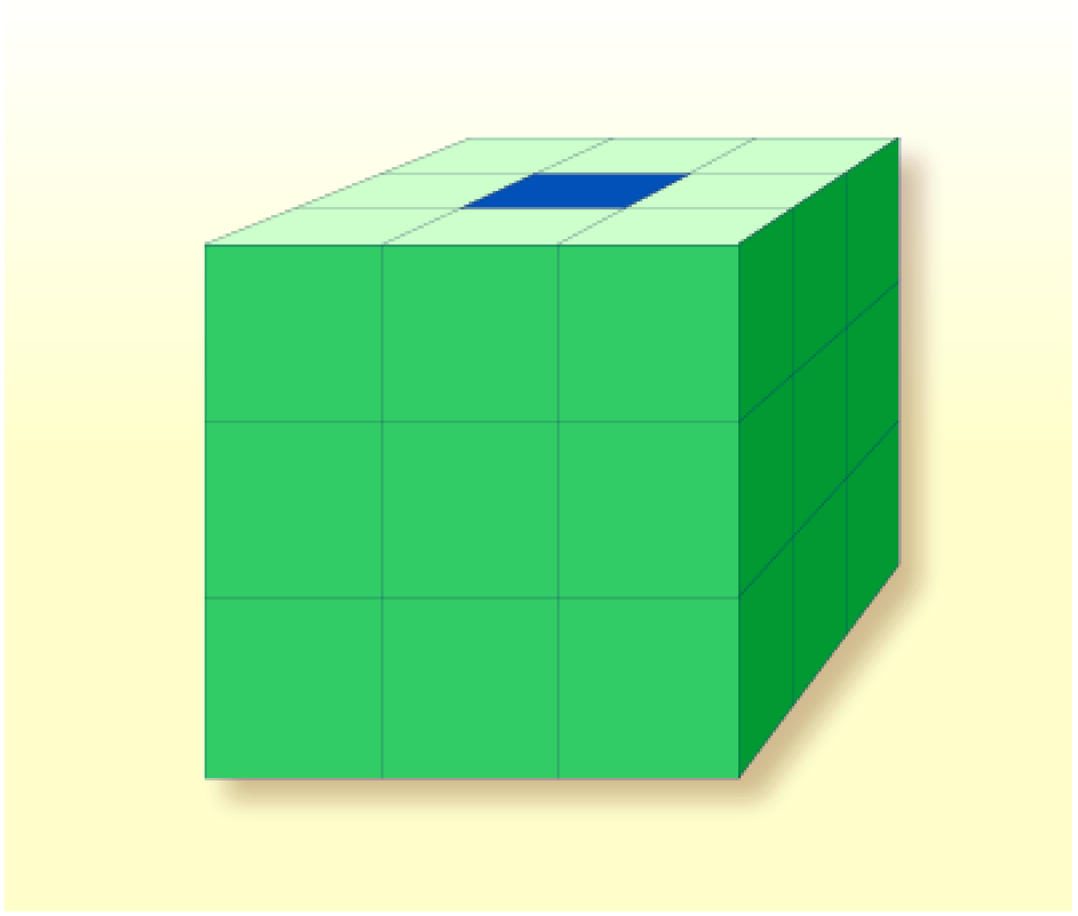
	B		
A	B		
	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (V,2) <'B',C',D'> (3,3) <'E'> (3,4) <'F'>
```

	B		
A	C		
	D	E	F

Dreidimensionales Array

Ein dreidimensionales Array könnte man sich folgendermaßen vorstellen:



Das oben dargestellte Array müsste wie folgt definiert werden (wobei gleichzeitig dem dunkel markierten Feld #FIELD2, das die Position Zeile 1, Spalte 2, Ebene 2 hat, ein Ausgangswert zugewiesen wird):

```
DEFINE DATA LOCAL
1 #ARRAY2
  2 #ROW (1:4)
    3 #COLUMN (1:3)
      4 #PLANE (1:3)
        5 #FIELD2 (P3) INIT (1,2,2) <100>
END-DEFINE
...
```

Wenn man das gleiche Array im Data-Area-Editor als Local Data Area definiert, sieht dies so aus:

```
I T L Name                            F Leng Index/Init/EM/Name/Comment
- - - -------------------------------- - ---- ---------------------------------
    1 #ARRAY2
    2 #ROW                                   (1:4)
    3 #COLUMN                                (1:3)
    4 #PLANE                                   (1:3)
I   5 #FIELD2                         P    3
```

Arrays als Teil einer größeren Datenstruktur

Die Mehrdimensionalität von Arrays ermöglicht es, Datenstrukturen analog zu COBOL- oder PL1-Strukturen zu definieren.

Beispiel:

```
DEFINE DATA LOCAL
1 #AREA
  2 #FIELD1 (A10)
  2 #GROUP1 (1:10)
    3 #FIELD2 (P2)
    3 #FIELD3 (N1/1:4)
END-DEFINE
...
```

Im obigen Beispiel hat der Datenbereich #AREA insgesamt eine Größe von:

10 + (10 * (2 + (1 * 4))) Bytes = 70 Bytes

#FIELD1 ist alphanumerisch und 10 Bytes lang. #GROUP1 ist ein Unterbereich von #AREA, hat 10 Ausprägungen und besteht aus zwei Feldern: #FIELD2 und #FIELD3. #FIELD2 ist gepackt numerisch und 2 Bytes lang; #FIELD3 ist das zweite Feld von #GROUP1, ist numerisch, 1 Byte lang und hat 4 Ausprägungen.

Wollen Sie eine bestimmte Ausprägung von #FIELD3 referenzieren, sind hierzu zwei Angaben erforderlich: erstens die der betreffenden Ausprägung von #GROUP1 und zweitens die der betreffenden Ausprägung von #FIELD3. Falls #FIELD3 beispielsweise an anderer Stelle im Programm in einem ADD-Statement referenziert würde, sähe dies folgendermaßen aus:

```
ADD 2 TO #FIELD3 (3,2)
```

Datenbank-Arrays

Adabas unterstützt Array-Strukturen innerhalb einer Datenbank in Form von **multiplen Feldern** und **Periodengruppen**. Diese sind im Abschnitt ***Datenbank-Arrays*** beschrieben.

Das folgende Beispiel zeigt einen `DEFINE DATA`-View, der ein multiples Feld enthält, zunächst programmintern und dann in einer programmexternen Local Data Area definiert:

```
DEFINE DATA LOCAL
1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE (1:10) /* <--- MULTIPLE-VALUE FIELD
END-DEFINE
...
```

Dieselbe View in einer programmexternen Local Data Area:

```
I T L Name                             F Leng Index/Init/EM/Name/Comment
- - - -------------------------------- - ---- ---------------------------------
  V 1 EMPLOYEES-VIEW                          EMPLOYEES
    2 NAME                             A   20
  M 2 ADDRESS-LINE                     A   20 (1:10) /* MU-FIELD
```

Arithmetische Ausdrücke in Index-Notationen

Zur Bestimmung eines Bereiches von Ausprägungen in einem Array können auch einfache arithmetische Ausdrücke verwendet werden.

Beispiele:

`MA (I:I+5)`	6 Werte des Feldes `MA` werden referenziert, beginnend mit Wert `I` und endend mit Wert `I + 5`.
`MA (I+2:J-3)`	Die Werte des Feldes `MA` von `I + 2` bis `J - 3` werden referenziert.

In derartigen Index-Angaben dürfen keine anderen Rechenzeichen als + und − verwendet werden.

Arithmetische Funktionen bei Arrays

Arithmetische Funktionen lassen sich innerhalb von Arrays auf Tabellenebene, auf Zeilen-/Spaltenebene und auf Feldebene einsetzen.

Allerdings sind mit Array-Variablen nur einfache arithmetische Funktionen erlaubt, die höchstens ein oder zwei Operanden enthalten sowie möglicherweise eine dritte Variable als Ergebnisfeld.

Werden Indexbereiche definiert, so sind nur die Operatoren + und − zulässig.

Beispiele für Array-Arithmetik:

Die folgenden Beispiele gehen von den folgenden Felddefinitionen aus:

```
DEFINE DATA LOCAL
01 #A (N5/1:10,1:10)
01 #B (N5/1:10,1:10)
01 #C (N5)
END-DEFINE
...
```

1. `ADD #A(*,*) TO #B(*,*)`

   Der Ergebnisoperand, Array `#B`, enthält die elementweise Addition des Arrays `#A` und des ursprünglichen Werts von Array `#B`.

2. `ADD 4 TO #A(*,2)`

   Die zweite Spalte des Arrays `#A` wird durch den ursprünglichen Wert plus 4 ersetzt.

3. `ADD 2 TO #A(2,*)`

   Die zweite Zeile des Arrays `#A` wird durch den ursprünglichen Wert plus 2 ersetzt.

4. `ADD #A(2,*) TO #B(4,*)`

   Der Wert der zweiten Zeile des Arrays `#A` wird zur vierten Zeile des Arrays `#B` addiert.

5. `ADD #A(2,*) TO #B(*,2)`

   Diese Operation ist nicht erlaubt und würde von Natural als Syntaxfehler zurückgewiesen. Es ist nicht gestattet, bei arithmetischen Funktionen Zeilen mit Spalten zu vermischen.

6. `ADD #A(2,*) TO #C`

   Alle Werte der zweiten Zeile des Arrays #A werden zu dem Skalarwert `#C` addiert.

7. `ADD #A(2,5:7) TO #C`

   Die Werte der 5. bis 7. Spalte der zweiten Zeile des Arrays `#A` werden zum Skalarwert `#C` addiert.

25 X-Arrays

- Definition 180
- Speicherverwaltung von X-Arrays 181
- Speicherverwaltung von X-Gruppen-Arrays 181
- X-Array referenzieren 183
- Parameter-Übertragung mit X-Arrays 184
- Parameter-Übertragung mit X-Group-Arrays 185
- X-Array mit dynamischen Variablen 187
- Unter- und Obergrenze eines Arrays 187

Wenn Sie ein normales Array definieren, müssen sie die Indexgrenzen und folglich die Anzahl der Ausprägungen für jede Dimension genau angeben. Zur Laufzeit existiert standardmäßig das vollständige Array-Feld; auf jede seiner Ausprägungen kann ohne zusätzliche Zuweisungsoperationen zugegriffen werden. Das Größen-Layout kann nicht mehr geändert werden; Sie können Feldausprägungen weder hinzufügen noch entfernen.

Zur Anwendungsentwicklungszeit kennen Sie wahrscheinlich nicht die genaue Anzahl der Ausprägungen eines Arrays. Sie möchten vielleicht aber die Größe eines Arrays zur Laufzeit ändern können.

Zu diesem Zweck können Sie ein sogenanntes X-Array (*eXtensible array* = erweiterbares Array) definieren. Die Größe eines X-Arrays kann zur Laufzeit geändert werden, wodurch Sie darin unterstützt werden, Ihren Hauptspeicher effizienter zu verwalten. Sie können z.B. eine große Anzahl von Array-Ausprägungen für einen kurzen Zeitraum benutzen und dann den Hauptspeicherplatz reduzieren, wenn die Anwendung den Array nicht mehr benutzt.

Definition

Ein X-Array ist ein Array, dessen Anzahl an Ausprägungen zur Kompilierungszeit nicht bekannt ist. Ein X-Array kann nur in einem `DEFINE DATA`-Statement definiert werden, wenn Sie einen Stern (*) für mindestens eine Grenze von wenigstens einer Dimension des Arrays angeben. Der Stern (*) in der Grenzen-Definition verweist darauf, dass die entsprechende Grenze verlängerbar ist. Nur eine Grenze — entweder die obere oder die untere — kann verlängert werden, aber nicht beide.

Ein X-Array kann immer dann definiert werden, wenn ein (fester) Array definiert werden kann, d.h. auf jeder Ebene oder sogar als indizierte Gruppe. Es kann nicht verwendet werden, um auf MU-PE-Felder zuzugreifen. Ein mehrdimensionales Array kann eine Mischung aus konstanten und verlängerbaren Grenzen haben.

Beispiel:

```
DEFINE DATA LOCAL
1 #X-ARR1 (A5/1:*)          /* lower bound is fixed at 1, upper bound is variable
1 #X-ARR2 (A5/*)            /* shortcut for (A5/1:*)
1 #X-ARR3 (A5/*:100)        /* lower bound is variable, upper bound is fixed at 100
1 #X-ARR4 (A5/1:10,1:*)     /* 1st dimension has a fixed index range with (1:10)
END-DEFINE                  /* 2nd dimension has fixed lower bound 1 and variable
upper bound
```

Speicherverwaltung von X-Arrays

Die Ausprägungen eines X-Arrays müssen ausdrücklich zugewiesen werden, bevor auf sie zugegriffen werden kann. Sie können für die Statements `EXPAND`, `RESIZE` und `REDUCE` die Anzahl der Ausprägungen für jede Dimension ändern.

Die Anzahl der Dimension des X-Arrays (1, 2 oder 3 Dimensionen) kann aber nicht geändert werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #X-ARR(I4/10:*)
END-DEFINE
EXPAND ARRAY #X-ARR TO (10:10000)
/* #X-ARR(10) to #X-ARR(10000) are accessible
WRITE *LBOUND(#X-ARR)                  /* is 10
   *UBOUND(#X-ARR)                     /* is 10000
   *OCCURRENCE(#X-ARR)                 /* is 9991
#X-ARR(*) := 4711                      /* same as #X-ARR(10:10000) := 4711
/* resize array from current lower bound=10 to upper bound =1000
RESIZE ARRAY #X-ARR TO (*:1000)
/* #X-ARR(10) to #X-ARR(1000) are accessible
/* #X-ARR(1001) to #X-ARR(10000) are released
WRITE *LBOUND(#X-ARR)                  /* is 10
   *UBOUND(#X-ARR)                     /* is 1000
   *OCCURRENCE(#X-ARR)                 /* is 991
/* release all occurrences
REDUCE ARRAY #X-ARR TO  0
WRITE *OCCURRENCE(#X-ARR)              /* is 0
```

Speicherverwaltung von X-Gruppen-Arrays

Wenn Sie Ausprägungen der X-Gruppen-Arrays erhöhen oder reduzieren möchten, müssen Sie zwischen unabhängigen und abhängigen Dimensionen unterscheiden.

Eine Dimension, die direkt (nicht weitergegeben) für einen X-Gruppen-Array angegeben wird, ist *unabhängig*.

Eine Dimension, die nicht *direkt* für einen X-Gruppen-Array angegeben, sondern weitergegeben wird, ist *abhängig*.

Nur die unabhängigen Dimensionen können in den Statements EXPAND, RESIZE und REDUCE geändert werden. Die Dimensionen müssen unter Verwendung des entsprechenden Namens des X-Gruppen-Arrays, zu dem diese Dimension als unabhängige Dimension gehört, geändert werden.

Beispiel - Unabhängige/abhängige Dimensionen:

```
DEFINE  DATA LOCAL
1 #X-GROUP-ARR1(1:*)                 /* (1:*)
  2 #X-ARR1    (I4)                  /* (1:*)
  2 #X-ARR2    (I4/2:*)              /* (1:*,2:*)
  2 #X-GROUP-ARR2                    /* (1:*)
    3 #X-ARR3  (I4)                  /* (1:*)
    3 #X-ARR4  (I4/3:*)              /* (1:*,3:*)
    3 #X-ARR5  (I4/4:*, 5:*)         /* (1:*,4:*,5:*)
END-DEFINE
```

Die folgende Tabelle zeigt, ob die Dimensionen in dem obengezeigten Programm unabhängig oder abhängig sind.

Name	Abhängige Dimension	Unabhängige Dimension
#X-GROUP-ARR1		(1:*)
#X-ARR1	(1:*)	
#X-ARR2	(1:*)	(2:*)
#X-GROUP-ARR2	(1:*)	
#X-ARR3	(1:*)	
#X-ARR4	(1:*)	(3:*)
#X-ARR5	(1:*)	(4:*,5:*)

Als Index-Notation für die abhängige Dimension ist entweder ein einzelner Stern (*), ein mit einem Stern definierter Bereich oder die Definition der Ober- und Untergrenze zulässig. Diese Angabe soll darauf verweisen, dass die Grenzen der abhängigen Dimension so bleiben müssen wie sie sind und nicht geändert werden können.

Die Ausprägungen der abhängigen Dimensionen können nur durch Manipulation der entsprechenden Array-Gruppen geändert werden.

```
EXPAND ARRAY #X-GROUP-ARR1 TO (1:11)       /* #X-ARR1(1:11) are allocated
                                           /* #X-ARR3(1:11) are allocated
EXPAND ARRAY #X-ARR2 TO (*:*, 2:12)        /* #X-ARR2(1:11, 2:12) are allocated
EXPAND ARRAY #X-ARR2 TO (1:*, 2:12)        /* same as before
EXPAND ARRAY #X-ARR2 TO (*  , 2:12)        /* same as before
EXPAND ARRAY #X-ARR4 TO (*:*, 3:13)        /* #X-ARR4(1:11, 3:13) are allocated
EXPAND ARRAY #X-ARR5 TO (*:*, 4:14, 5:15)  /* #X-ARR5(1:11, 4:14, 5:15) are allocated
```

Die EXPAND-Statements können in beliebiger Reihenfolge kodiert werden.

Die folgende Verwendung des EXPAND-Statements ist nicht zulässig, da die Arrays nur abhängige Dimensionen haben.

```
EXPAND ARRAY #X-ARR1 TO ...
EXPAND ARRAY #X-GROUP-ARR2 TO ...
EXPAND ARRAY #X-ARR3 TO ...
```

X-Array referenzieren

Die Ausprägungen eines X-Arrays müssen über ein EXPAND- oder RESIZE-Statement zugewiesen werden, bevor sie aufgerufen werden können. Die Statements READ, FIND und GET ordnen Ausprägungen implizit zu, wenn die Werte von einer Tamino-Datenbank stammen.

In der Regel gilt: der Versuch, eine nicht existierende X-Array-Ausprägung zu adressieren führt zu einem Laufzeitfehler. Bei einigen Statements verursacht der Zugriff auf ein nicht verwirklichtes X-Array-Feld jedoch keinen Fehler, wenn alle Ausprägungen eines X-Arrays mit der kompletten Bereichsnotation referenziert werden, zum Beispiel #X-ARR(*). Dies gilt für

- Parameter, die in einem CALL-Statement benutzt werden,
- Parameter, die bei CALLNAT, PERFORM, SEND EVENT oder OPEN DIALOG benutzt werden, wenn sie als OPTIONAL definiert sind,
- Source-Felder, die in einem COMPRESS-Statement benutzt werden,
- Ausgabefelder, die mit einem PRINT-Statement angegeben werden,
- Felder, die in einem RESET-Statement referenziert werden.

Wenn individuelle Ausprägungen eines nicht verwirklichten X-Arrays in einem dieser Statements referenziert werden, wird eine entsprechende Fehlermeldung ausgegeben.

Beispiel:

```
DEFINE DATA LOCAL
1 #X-ARR (A10/1:*) /* X-array only defined, but not allocated
END-DEFINE
RESET #X-ARR(*)    /* no error, because complete field referenced with (*)
RESET #X-ARR(1:3)  /* runtime error, because individual occurrences (1:3) are
referenced
END
```

Die Stern-Notation (*) in einer Array-Referenz steht für den kompletten Bereich einer Dimension. Wenn das Array ein X-Array ist, dann steht der Stern für den Indexbereich der aktuell zugewiesenen Unter- und Obergrenze, die mit `*LBOUND` und `*UBOUND` festgelegt wird.

Parameter-Übertragung mit X-Arrays

Als Parameter benutzte X-Arrays werden im Hinblick auf die Überprüfung folgender Elemente wie Konstanten-Arrays behandelt:

- Format
- Länge
- Dimension

  oder

- Anzahl der Ausprägungen

Außerdem können X-Array-Parameter auch die Anzahl der Ausprägungen ändern, wenn Sie die Statements `RESIZE`, `REDUCE` oder `EXPAND` benutzen. Die RESIZE-Funktion eines X-Array-Parameters ist von drei Faktoren abhängig:

- der Art der benutzten Parameter-Übertragung, d.h. By Reference oder By Value
- der Definition des Caller oder des X-Array-Parameters
- dem Typ des weitergegebenen X-Array-Bereichs (kompletter Bereich oder Unterbereich)

Die folgenden Tabellen zeigen, wann ein `EXPAND`, `RESIZE` oder `REDUCE` eines X-Array-Parameters zulässig ist.

Beispiel mit CALL By Value

CALLER	PARAMETER		
	Statisch	Variable (1:V)	`X-Array`
Statisch	Nein	Nein	Ja
X-Array, Unterbereich, z.B. `CALLNAT...#XA(1:5)`	Nein	Nein	Ja
X-Array, kompletter Bereich, z.B. `CALLNAT...#XA(*)`	Nein	Nein	Ja

CALL By Reference/CALL By Value Result

<table>
<tr><th rowspan="2">CALLER</th><th colspan="4">PARAMETER</th></tr>
<tr><td>Statisch</td><td>Variable (1:V)</td><td>X-Array mit einer festen Untergrenze, kompletter Bereich, z.B.
<code>DEFINE DATA PARAMETER
1 #PX (A10/1:*)</code></td><td>X-Array mit einer festen Obergrenze, kompletter Bereich, z.B.
<code>DEFINE DATA PARAMETER
1 #PX (A10/*:1)</code></td></tr>
<tr><td>Statisch</td><td>Nein</td><td>Nein</td><td>Nein</td><td>Nein</td></tr>
<tr><td>X-Array, Unterbereich, z.B.
<code>CALLNAT...#XA(1:5)</code></td><td>Nein</td><td>Nein</td><td>Nein</td><td>Nein</td></tr>
<tr><td>X-Array mit einer festen Untergrenze, kompletter Bereich, z.B.
<code>DEFINE DATA LOCAL
1 #XA(A10/1:*)
...
CALLNAT...#XA(*)</code></td><td>Nein</td><td>Nein</td><td>Ja</td><td>Nein</td></tr>
<tr><td>X-Array mit einer festen Obergrenze, kompletter Bereich, z.B.
<code>DEFINE DATA LOCAL
1 #XA(A10/*:1)
...
CALLNAT...#XA(*)</code></td><td>Nein</td><td>Nein</td><td>Nein</td><td>Ja</td></tr>
</table>

Parameter-Übertragung mit X-Group-Arrays

Die Deklaration eines X-Group-Arrays impliziert, dass jedes Element der Gruppe dieselben Werte für die Ober- und Untergrenze hat. Aus diesem Grund kann die Anzahl der Ausprägungen von abhängigen Felddimensionen eines X-Group-Arrays nur geändert werden, wenn der Gruppenname des X-Group-Arrays mit dem Statement `RESIZE`, `REDUCE` und `EXPAND` angegeben wird (siehe *Speicherverwaltung von X-Gruppen-Arrays* oben).

Bestandteile von X-Group-Arrays können als Parameter an X-Group-Arrays übertragen werden, die in einer Parameter Data Area definiert sind. Die Gruppenstrukturen des Aufrufers und des Aufgerufenen müssen nicht unbedingt identisch sein. Ein `RESIZE`, `REDUCE` und `EXPAND`, das vom Aufgerufenen gemacht wird, ist nur möglich solange das X-Group-Array des Aufrufers gleich bleibt.

Beispiel - Elemente eines X-Group Array als Parameter übertragen:

Programm:

```
DEFINE DATA LOCAL
1 #X-GROUP-ARR1(1:*)                    /* (1:*)
   2 #X-ARR1 (I4)                       /* (1:*)
   2 #X-ARR2 (I4)                       /* (1:*)
1 #X-GROUP-ARR2(1:*)                    /* (1:*)
   2 #X-ARR3 (I4)                       /* (1:*)
   2 #X-ARR4 (I4)                       /* (1:*)
END-DEFINE
...
CALLNAT ... #X-ARR1(*) #X-ARR4(*)
...
END
```

Subprogramm:

```
DEFINE DATA PARAMETER
1 #X-GROUP-ARR(1:*)                     /* (1:*)
   2 #X-PAR1 (I4)                       /* (1:*)
   2 #X-PAR2 (I4)                       /* (1:*)
END-DEFINE
...
RESIZE ARRAY #X-GROUP-ARR to (1:5)
...
END
```

Das `RESIZE`-Statement ist im Subprogramm nicht möglich. Das Ergebnis wäre eine uneinheitliche Anzahl von Ausprägungen der Felder, die in den X-Group-Arrays des Programms definiert sind.

X-Array mit dynamischen Variablen

Ein X-Array mit dynamischen Variablen kann zugewiesen werden, indem man zuerst die Anzahl der Ausprägungen mit Hilfe des EXPAND-Statements angibt und dann den zuvor zugewiesenen Array-Ausprägungen einen Wert zuweist.

Beispiel:

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
EXPAND  ARRAY   #X-ARRAY TO (1:10)
  /* allocate #X-ARRAY(1) to #X-ARRAY(10) with zero length.
  /* *LENGTH(#X-ARRAY(1:10)) is zero
#X-ARRAY(*) := 'abc'
  /* #X-ARRAY(1:10) contains 'abc',
  /* *LENGTH(#X-ARRAY(1:10)) is 3
EXPAND  ARRAY   #X-ARRAY TO (1:20)
  /* allocate #X-ARRAY(11) to #X-ARRAY(20) with zero length
  /* *LENGTH(#X-ARRAY(11:20)) is zero
#X-ARRAY(11:20) := 'def'
  /* #X-ARRAY(11:20) contains 'def'
  /* *LENGTH(#X-ARRAY(11:20)) is 3
```

Unter- und Obergrenze eines Arrays

Die Systemvariablen *LBOUND und *UBOUND enthalten die aktuelle Unter- und Obergrenze eines Arrays für die angegebene(n) Dimension(en) (1, 2 oder 3).

Wenn keine Ausprägungen eines X-Arrays zugewiesen worden sind, ist der Aufruf von *LBOUND oder *UBOUND für die variablen Indexgrenzen nicht definiert, d.h. für die durch einen Stern (*) in der Indexdefinition dargestellten Grenzen, und führt zu einem Laufzeitfehler. Um einen Laufzeitfehler zu vermeiden, kann *OCCURRENCE benutzt werden, um auf Null-Ausprägungen zu überprüfen, bevor *LBOUND oder *UBOUND ausgewertet wird:

Beispiel:

```
IF *OCCURRENCE (#A) NE 0  AND  *UBOUND(#A) < 100 THEN ...
```

26 Benutzer-definierte Funktionen

- Einführung in benutzer-definierte Funktionen 190
- Unterschied zwischen Function Call und Subprogram Call 190
- Definition einer Function (DEFINE FUNCTION) 192
- Definition eines Prototype (DEFINE PROTOTYPE) 192
- Symbolischer und variabler Function Call 193
- Automatische/Implizite Prototype-Definition (APT) 193
- Prototype Cast (PT-Klausel) 193
- Zwischenergebnis für den Rückgabewert (IR-Klausel) 194
- Kombinationen möglicher Prototype-Definitionen 194
- Rekursiver Function Call 196
- Behavior of Functions in Statements and Expressions 197
- Usage of Functions as Statements 198

Einführung in benutzer-definierte Funktionen

Funktionen (Functions) bieten Ihnen ähnlich wie Subprogramme die Möglichkeit, Daten zu empfangen, diese zu verändern und die Ergebnisse an das rufende Modul zurückzugeben. Der Vorteil, den Funktionen im Vergleich zu Subprogrammen bieten, liegt darin, dass Funktionsaufrufe (Function Calls) direkt innerhalb von Statements und Ausdrücken verwendet werden können, ohne dass dazu zusätzliche temporäre Variablen nötig sind.

Normalerweise wird je nach den Parametern, die der Function beigegeben werden, das Ergebnis in der Function erzeugt und wird dann an das rufende Objekt zurückgegeben. Falls andere Werte an das rufende Modul zurückgegeben werden sollen, kann dies mit Hilfe der Parameter erfolgen; siehe ***Subprogramm***.

Sobald der Code der Function vollständig ausgeführt worden ist, wird die Kontrolle an das rufende Objekt zurückgegeben und das Programm fährt mit dem nach dem Function Call vorhandenen Statement fort.

Weitere Informationen:

- Natural-Objekttyp **Function**
- **Function Call**
- Natural-Statements `DEFINE FUNCTION`, `DEFINE PROTOTYPE`

Unterschied zwischen Function Call und Subprogram Call

Das folgende Beispiel veranschaulicht den Unterschied zwischen der Verwendung eines Function Call und eines Subprogram Call.

Beispiel für die Verwendung eines Function Call:

The following example comprises a program object that uses a function call, a function object containing a function definition created with a `DEFINE FUNCTION` statement, and a copycode object created with a `DEFINE PROTOTYPE` statement.

Programm-Objekt:

```
/* Excerpt from a Natural program using a function call
INCLUDE C#ADD
WRITE #ADD(< 2,3 >) /* function call; no temporary variable necessary
END
```

Function-Objekt:

```
/* Natural function definition
DEFINE FUNCTION #ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #ADD := #SUMMAND1 + #SUMMAND2
END-FUNCTION
END
```

Copycode-Objekt (zum Beispiel "C#ADD"):

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

Um die gleiche Funktionalität mit Hilfe eines Subprogramms zu erzielen, müssen Sie temporäre Variablen verwenden.

Beispiel für die Verwendung eines Subprogram Call:

Das folgende Beispiel enthält ein Objekt vom Typ Programm, das ein Objekt des Typs Subprogramm unter Verwendung einer temporären Variablen aufruft.

Programm-Objekt:

```
/* Natural program using a subprogram
DEFINE DATA LOCAL
1 #RESULT (I4) INIT <0>           /* temporary variable
END-DEFINE

CALLNAT 'N#ADD' USING #RESULT 2 3 /* result is stored into #RESULT
WRITE #RESULT                     /* print out the result of the subprogram
END
```

Subprogramm-Objekt (zum Beispiel "N#ADD"):

```
/* Natural program using a subprogram
DEFINE DATA PARAMETER
1 #RETURN   (I4) BY VALUE RESULT
1 #SUMMAND1 (I4) BY VALUE
1 #SUMMAND2 (I4) BY VALUE
END-DEFINE

#RETURN := #SUMMAND1 + #SUMMAND2
END
```

Definition einer Function (DEFINE FUNCTION)

Die Funktionsdefinition besteht aus dem Natural-Code, der ausgeführt werden soll, wenn die Function aufgerufen wird. Ähnlich wie bei Subprogrammen müssen Sie zunächst ein Natural-Objekt anlegen, in diesem Fall vom Typ „Function“, das die Funktionsdefinition enthält. Die Funktionsdefinition wird mit Hilfe des Natural-Statements `DEFINE FUNCTION` erstellt.

Bei dem Funktionsaufruf (**Function Call**) kann es sich um einen beliebigen Objekttyp handeln, der ausführbaren Code enthält.

Definition eines Prototype (DEFINE PROTOTYPE)

Um Function Calls kompilieren zu können, benötigt Natural Informationen über Format/Länge- und ggf. die Array-Definition des Rückgabewerts. Diese Informationen werden dann dem Compiler in der Prototype-Definition zur Verfügung gestellt. Die Prototype-Definition wird mit dem Natural-Statement `DEFINE PROTOTYPE` angelegt. Dort können Sie auch die Definition des zurückzugebenden Parameters einfügen, die dann bei der Kompilierung geprüft wird.

Da Natural die Verbindung zwischen rufenden und gerufenen Objekten zur Laufzeit und nicht vorher herstellt, weiss der Rechner nicht, mit welcher Art von Function-Rückgabewert er es bei der Kompilierung zu tun hat. Das liegt daran, dass das Objekt, das die Function enthält, nicht notwendigerweise (während des Kompilierens) existieren muss. Darum wird die Prototype-Definition angelegt, so dass die Format/Länge- und ggf. die Array-Definition beim Kompilieren in das generierte Programm (GP) hinein generiert werden kann.

Es ist wichtig zu wissen, dass eine Prototype-Definition niemals ausführbaren Code enthält. Eine Prototype-Definition enthält lediglich die folgenden Informationen über einen Function Call: die Format/Länge- und ggf. die Array-Definition des Rückgabewerts bzw. des Parameters, der zurückgereicht wird.

Symbolischer und variabler Function Call

Um einen variablen Function Call zu definieren müssen Sie immer ein `DEFINE PROTOTYPE VARIABLE`-Statement benutzen. Andernfalls wird angenommen, dass es sich bei dem Function Call um einen impliziten symbolischen Function Call handelt.

Weitere Informationen zu diesem Thema finden Sie im Abschnitt ***Function Call***.

Automatische/Implizite Prototype-Definition (APT)

Wenn weder eine explizite Prototype-Definition (EPT) noch eine `PT`-Klausel (**Prototype Cast**) existiert, findet im generierten Programm die Suche nach der Prototype-Definition statt. Weitere Informationen, siehe ***Kombinationen möglicher Prototype-Definitionen*** weiter unten.

Prototype Cast (PT-Klausel)

Um den entsprechenden Prototype einer bestimmten Function zu finden, sucht Natural nach einem Prototype, der den Namen der Function trägt. Falls ein solcher Prototype nicht gefunden wird, geht Natural davon aus, dass es ein symbolischer Function Call ist. In diesem Fall muss die „Signatur“ der Function unter Verwendung des Schlüsselworts `PT=` in dem Function Call definiert werden.

Zwischenergebnis für den Rückgabewert (IR-Klausel)

Mit dieser Klausel können Sie den Rückgabewert für einen Function Call angeben, ohne eine explizite oder implizite Prototype-Definition zu verwenden, das heisst, sie ermöglicht die explizite Angabe eines Zwischenergebnisses. Weitere Informationen siehe ***Function Call***, *`intermediate-result-definition`*

Kombinationen möglicher Prototype-Definitionen

Die folgende Tabelle verdeutlicht die Auswirkungen auf eine Prototype-Definition in Abhängigkeit von verschiedenen Syntax-Kombinationen, die bei der Benutzung des `DEFINE PROTOTYPE`-Statements und/oder der im Function Call vorhandenen Klauseln möglich sind. Folgende Möglichkeiten gibt es, um Teile eines Function Prototype zu definieren, die sich nur auf den Function Call auswirken, zu dem sie gehören:

- **Explizite DEFINE PROTOTYPE-Definition (EPT)**
  Kann symbolischen/variablen Function Call, Parameter-Definition, Rückgabewert-Definition festlegen.
- **Prototype Cast (`PT`-Klausel)**
  Kann Parameter-Definition, Rückgabewert-Definition festlegen.
- **Zwischenergebnis für den Rückgabewert (IR-Klausel)**
  Kann Rückgabewert-Definition festlegen.

Fall	Explizite Prototype-Definition in `DEFINE PROTOTYPE` (EPT)	`PT`-Klausel im Function Call (PT)	`IR`-Klausel im Function Call (IR)		Automatisches Einlesen der Prototype-Definition aus dem GP (APT)	Prototype-Verhalten
1	x	x	x		-	SV(EPT), PS(PT), R(IR)
2	-	x	x		-	S, PS(PT), R(IR)
3	x	-	x		-	SV(EPT), PS(EPT), R(IR)
4	-	-	x		x	S, PS(APT), R(IR)
5	x	x	-		-	SV (EPT), PS(PT), R(PT)
6	-	x	-		-	S, PS(PT), R(IR)
7	x	-	-		-	SV(EPT), PS(EPT), R(EPT)
8	-	-	-		x	S, PS(APT), R(APT)

Dabei ist:

EPT	Explizites DEFINE PROTOTYPE-Statement.
PT	Prototype Cast (PT-Klausel).
IR	Zwischenergebnis für den Rückgabewert(IR-Klausel).
APT	Automatische Prototype-Definition über externes GP.
S	Symbolischer Function Call.
V	Variabler Function Call.
SV(EPT)	Explizite Prototype-Definition entscheidet, ob ein symbolischer oder ein variabler Function Call durchgeführt wird.
R(IR)	Die Rückgabevariable (R) wird durch die IR-Klausel im Function Call definiert.
R(PT)	Die Rückgabevariable (R) wird durch die PT-Klausel im Function Call definiert.
R(EPT)	Die Rückgabevariable (R) wird durch das explizite DEFINE PROTOTYPE-Statement definiert.
PS(PT)	Die Parameter-Signatur (PS) (d.h. die Parameter-Definition ohne Rückgabewert-Definition) wird durch die PT-Klausel im Function Call definiert.
PS(EPT)	Die Parameter-Signatur (PS) (d.h. die Parameter-Definition ohne Rückgabewert-Definition) wird durch das explizite DEFINE PROTOTYPE-Statement definiert.
PS(APT)	Die Parameter-Signatur (PS) wird automatisch durch das Einlesen in die Prototype-Definition aus dem generierten Programm (GP) definiert.
R(APT)	Die Rückgabevariable (R) wird durch automatische Prototype-Definition über externes GP definiert.

Als Beispiel das Verhalten gemäß Fall 1 aus der obigen Tabelle:

Wie ist das Verhalten, wenn ein explizites DEFINE PROTOTYPE-Statement (EPT) verwendet wird und wenn im Function Call die PT- und IR-Klauseln definiert sind?

Die EPT-Definition entscheidet, ob ein symbolischer oder ein variabler Function Call durchgeführt wird. Von einem variablen Function Call wird ausgegangen, wenn zuvor ein DEFINE PROTOTYPE VARIABLE-Statement definiert worden ist. Die Parameter-Signatur (d.h. die Format/Länge-Definition aller Parameter ohne Rückgabewert-Definition) wird durch die PT-Klausel definiert, und die Format/Länge-Angabe des Rückgabewerts wird durch die IR-Klausel im Function Call festgelegt. In diesem Fall wird keine automatische Prototype-Definition (APT) gestartet.

Abschließend können aus den oben aufgeführten Fällen die folgenden allgemeinen Regeln abgeleitet werden:

- Im Falle von variablen Function Calls muss für den Aufruf immer eine explizite Prototype-Definition (ETP) vorhanden sein.
- Die PT-Klausel entscheidet nicht, ob es sich um einen symbolischen oder einen variablen Function Call handelt.
- Die Definitionen in der PT-Klausel überschreiben die ETP-Definitionen für Parameter und Rückgabewert.

- Die Definitionen in der `IR`-Klausel überschreiben die Rückgabewert-Defintion.
- Wenn weder eine ETP-Definition noch eine `PT`-Klausel existiert, erfolgt im generierten Program die Suche nach der Prototyp-Definition (automatische Prototyp-Definition).

Rekursiver Function Call

Wenn die Funktion rekursiv aufgerufen werden soll, muss der Prototyp der Function in der Function-Definition enthalten sein oder er muss mittels einer `INCLUDE`-Datei eingefügt werden.

Beispiel:

Function-Objekt:

```
/* Function definition for calculation of the math. factorial
DEFINE FUNCTION #FACT
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  LOCAL
  1 #TEMP (I4)
  END-DEFINE

  /* Prototype definition is necessary
  INCLUDE C#FACT

  /* Program code
  IF #PARA=0
    #FACT := 1
  ELSE
    #TEMP := #PARA - 1
    #FACT := #PARA * #FACT(< #TEMP >)
  END-IF

END-FUNCTION
END
```

Copycode-Objekt (zum Beispiel mit dem Namen C#FACT):

```
/* Prototype definition is necessary
DEFINE PROTOTYPE #FACT
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

Programm-Objekt:

```
/* Prototype definition
INCLUDE C#FACT

/* function call
WRITE #FACT(<12>)
END
```

Behavior of Functions in Statements and Expressions

Instead of operands, function calls can be used directly in statements or expressions. However, this is only allowed in places where operands cannot be modified.

All function calls are executed according to their syntactical sequence which is analyzed at compile time. The results of the function calls are saved in internal temporary variables and passed to the statement or expression.

This fixed sequence makes it possible to allow and execute standard output in functions, without, for example, unwillingly influencing the output of a statement.

Example:

Program:

```
/* Natural program using a function call
INCLUDE CPRINT
PRINT 'before' #PRINT(<>) 'after'
END
```

Function Object:

```
/* Natural function definition
/* function returns integer value 10
DEFINE FUNCTION #PRINT
  RETURNS (I4)
  WRITE '#PRINT'
  #PRINT := 10
END-FUNCTION
END
```

Copycode (for example, "CPRINT"):

```
DEFINE PROTOTYPE #PRINT END-PROTOTYPE
```

The following is the result which is then sent to the standard output:

```
#PRINT
before       10 after
```

Usage of Functions as Statements

Functions can also be called as statements independently from statements and expressions. In this case, the return value - assuming it has been defined - is not taken into account.

If, however, an independent function is declared after an optional operand list, the operand list must be followed by a semicolon to make it clear that the function call is not a part of the operand list.

Example:

Program Object:

```
/* Natural program using a function call
DEFINE DATA LOCAL
1 #A (I4) INIT <1>
1 #B (I4) INIT <2>
END-DEFINE

INCLUDE CPROTO

WRITE #A #B
```

```
#PRINT_ADD(< 2,3 >) /* function call belongs to operand list just in front of it

WRITE '******'

WRITE #A #B;        /* semicolon separates operand list and function call
#PRINT_ADD(< 2,3 >) /* function call doesn't belong to the operand list
END
```

Function Object:

```
/* Natural function definition
DEFINE FUNCTION #PRINT_ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #PRINT_ADD := #SUMMAND1 + #SUMMAND2
  PRINT '#PRINT_ADD =' #PRINT_ADD
END-FUNCTION
END
```

Copycode Object (for example, named `CPROTO`):

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #PRINT_ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

27 Datenbankzugriffe

Dieser Teil beschreibt verschiedene Aspekte des Zugriffs auf Daten in einer Datenbank mit Natural.

- **Natural und Datenbankzugriff**
- **Daten in einer Adabas-Datenbank aufrufen**
- **Daten in einer SQL-Datenbank aufrufen** (Dieser Abschnitt liegt nur in englischer Sprache vor.)
- **Accessing Data in a Tamino Database** (Dieser Abschnitt liegt nur in englischer Sprache vor.)

28 Natural und Datenbankzugriff

- Von Natural unterstützte Datenbankverwaltungssysteme 204
- Profilparameter zur Beeinflussung der Datenbankzugriffe 205
- Zugriff über Datendefinitionsmodule 205
- Eingebaute Datenmanipulationssprache 206
- Spezielle SQL-Statements in Natural 207

Dieses Kapitel gibt einen Überblick über die Funktionen, die Natural zum Zugriff auf verschiedene Typen von Datenbankverwaltungssysteme bietet.

Von Natural unterstützte Datenbankverwaltungssysteme

Natural bietet spezielle Datenbank-Schnittstellen für die folgenden Arten von Datenbankverwaltungssystemen (DBMS):

- Geschachtelte relationale DBMS (Adabas)
- DBMS vom Typ SQL (Oracle, Sybase, Informix, MS SQL Server)
- DBMS vom Typ XML (Tamino)

Die folgenden Themen werden im Folgenden erörtert:

- Adabas
- Tamino
- SQL-Datenbanken

Adabas

Über seine integrierte Adabas-Schnittstelle kann Natural auf Adabas-Datenbanken entweder auf einer lokalen Maschine oder auf entfernten Computern zugreifen. Beim entfernten Zugriff ist eine zusätzliche Weiterleitungs- und Kommunikationssoftware, wie z.B. Entire Net-Work, erforderlich. Auf jeden Fall ist die Art von Host-Maschine, auf der die Adabas-Datenbank läuft, für den Natural-Benutzer transparent.

Tamino

Natural for Tamino bietet die Möglichkeit, auf einen Tamino-Datenbank-Server auf einer lokalen Maschine oder, unter Verwendung eines nativen HTTP-Protokolls, auf einer entfernten Host-Maschine zuzugreifen. Der Zugriff auf eine Tamino-Datenbank kann auf die gleiche Art erfolgen wie bei Adabas- oder SQL-Datenbanken.

SQL-Datenbanken

Der Zugriff von Natural auf SQL-Datenbanken erfolgt über Entire Access, eine generische Schnittstellen- und Verteilungssoftware, die verschiedene SQL-Datenbankverwaltungssystemen wie z.B. Oracle, MS SQL Server oder standardisierte ODBC-Verbindungen unterstützt. Ausführliche Informationen über unterstützte SQL-Datenbankverwaltungssysteme und Plattformen finden Sie in der *Entire Access*-Dokumentation.

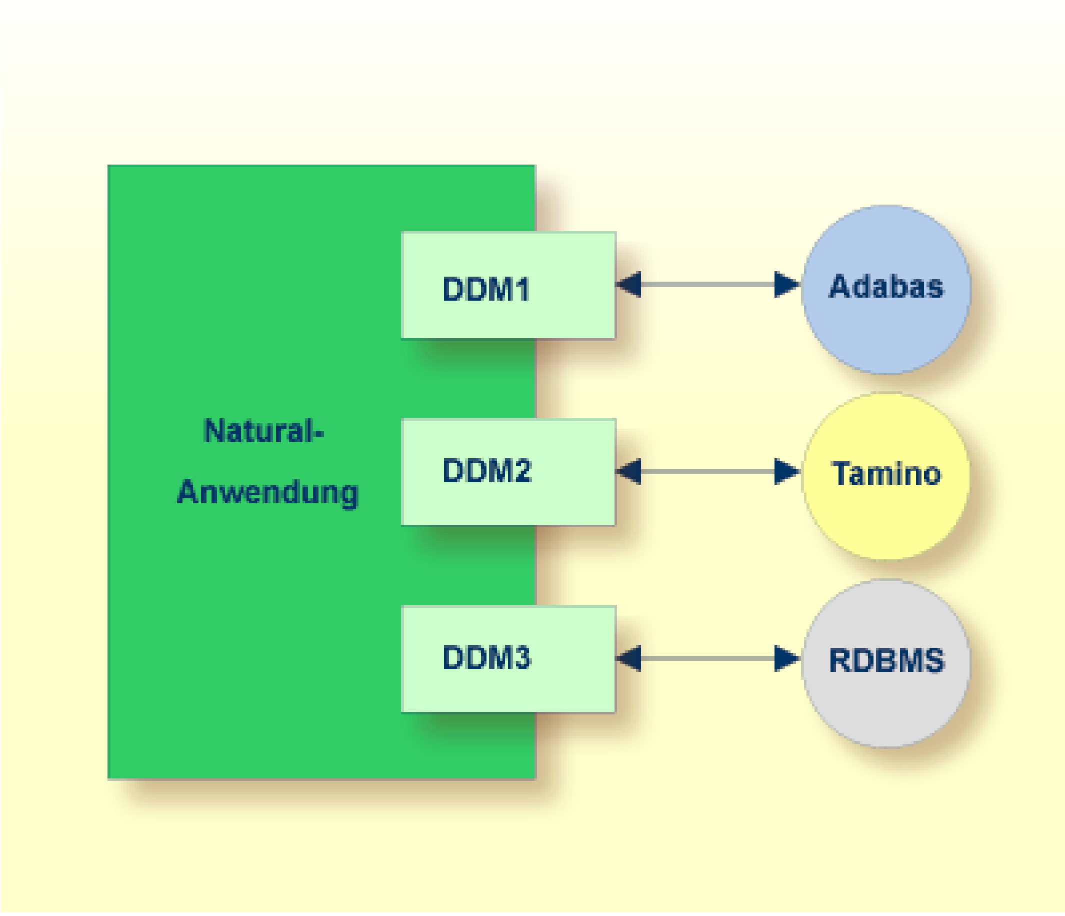
Profilparameter zur Beeinflussung der Datenbankzugriffe

Es gibt zahlreiche Profilparameter, die Einfluß darauf haben, wie Datenbankzugriffe in Natural behandelt werden. Eine Übersicht über diese Profilparameter finden Sie im Abschnitt *Database Management* in *Overview of Profile Parameters* in der *Configuration Utility*-Dokumentation. Eine ausführliche Beschreibung der dort aufgeführten Profilparameter finden Sie in den entsprechenden Abschnitten der *Parameter Reference*.

Zugriff über Datendefinitionsmodule

Um einen komfortablen und klaren Zugriff auf die verschiedenen Datenbankverwaltungssysteme zu ermöglichen, wird ein spezifisches Objekt, das Datendefinitionsmodul (Data Definition Module/DDM), in Natural benutzt. Dieses DDM stellt die Verbindung zwischen Natural-Datenstrukturen und den Datenstrukturen im zu benutzenden Datenbanksystem her. Eine solche Datenbankstruktur könnte eine Tabelle in einer SQL-Datenbank oder eine Datei in einer Adabas-Datenbank oder ein Dokumententyp (Doctype) in a Tamino-Datenbank sein. Folglich verbirgt das DDM die wahre Struktur der von der Natural-Anwendung aufgerufenen Datenbank. DDMs werden mit dem Natural DDM-Editor erstellt.

Natural kann von aus einer einzelnen Anwendung heraus auf verschiedene Arten von Datenbanken zugreifen (Adabas, Tamino, RDBMS), indem es Referenzen auf die DDMs benutzt, die die spezifischen Datenstrukturen im spezifischen Datenbanksystem darstellen. Die Abbildung weiter unten zeigt eine Anwendung, die eine Verbindung zu verschiedenen Arten von Datenbanken herstellt.



Eingebaute Datenmanipulationssprache

Natural hat eine eingebaute Datenmanipulationssprache (DML), die es Natural-Anwendungen ermöglicht, auf alle von Natural unterstützten Datenbanksysteme mittels derselben Sprach-Statements, wie z.B. `FIND`, `READ`, `STORE` oder `DELETE` zuzugreifen. Diese Statements können in einer Natural-Anwendung benutzt werden, ohne dass Sie die Art der Datenbank kennen, auf die gerade zugegriffen wird.

Natural ermittelt in seiner Konfigurationsdatei das Datenbanksystem und übersetzt die DMLStatements in datenbank-spezifische Kommandos; d.h. Natural generiert Direktkommandos für Adabas, SQL Statement-Strings und Hostvariablen-Strukturen für SQL-Datenbanken sowie XQuery-Abfragen für eine Tamino-Datenbank.

Da einige der Natural DML-Statements eine Funktionalität anbieten, die nicht für alle Datenbank-Arten unterstützt werden kann, ist die Benutzung dieser Funktionalität auf spezifische Datenbanksysteme beschränkt. Bitte beachten Sie die betreffenden datenbankspezifischen Erwägungen in der *Statements*-Dokumentation.

Spezielle SQL-Statements in Natural

Außer den „normalen" Natural DML-Statements bietet Natural eine Menge von SQLStatements für einen spezifischeren Einsatz in Verbindung mit SQL-Datenbanksystemen; siehe die *SQL Statements-Übersicht* in der *Statements*-Dokumentation).

Flexible SQL und Funktionen zum Verarbeiten von *Stored Procedures* vervollständigen die SQL-Kommandos. Diese Statements können nur für den Zugriff auf SQL-Datenbanken benutzt werden und gelten nicht für Adabas oder andere Nicht-SQL-Datenbanken.

29 Daten in einer Adabas-Datenbank aufrufen

- Adabas-Datenbankverwaltungsschnittstellen ADA und ADA2 210
- Datendefinitionsmodule (DDMs) 210
- Datenbank-Arrays 212
- Datenbank-View definieren 219
- Statements für Datenbankzugriffe 222
- Multi-Fetch-Klausel 235
- Datenbank-Verarbeitungsschleifen 236
- Datenänderungen - Transaktionsverarbeitung 243
- Datensätze mit ACCEPT/REJECT auswählen 251
- AT START/END OF DATA-Statements 255
- Unicode-Daten 257

Dieses Dokument beschreibt verschiedene Aspekte des Aufrufs von Daten in einer Adabas-Datenbank mit Natural.

Adabas-Datenbankverwaltungsschnittstellen ADA und ADA2

Bei den in Natural vorhandenen Adabas-Datenbankverwaltungsschnittstellentypen ADA und ADA2 handelt es sich um Schnittstellen, die voneinander so verschieden sind wie zum Beispiel die Typen ADA und SQL.

Der Typ ADA ist die Standardschnittstelle zu Adabas-Datenbanken. Sie kommt in Frage, wenn keine neue Adabas-Funktionalität zu berücksichtigen ist, die mit Adabas Version 6 für Open Systems und Adabas Version 8 für Großrechner eingeführt wurde.

Der Typ ADA2 ist eine erweiterte Schnittstelle zu Adabas-Datenbanken ab Adabas Version 6 für Open Systems und ab Adabas Version 8 für Großrechner. Sie unterstützt insbesondere Adabas-LA-Felder, große Adabas-Objektfelder und erweiterte Adabas-Puffer-Größen. Voraussetzung für die Unterstützung von Adabas-LA-Feldern und großen Objektfeldern ist die Verwendung des Natural-Formats `(A) DYNAMIC` in einer View-Definition. Die Unterstützung von erweiterten Adabas-Puffer-Größen ermöglicht die Definition von View-Größen von mehr als 64 KB. Weitere Informationen siehe ***Datenbank-View definieren***.

Der Datenbanktyp ADA2 unterstützt keine **Multi-Fetch-Verarbeitung**. Entsprechende globale und lokale Definitionen werden zur Laufzeit ignoriert.

Software AG-Produkte, die eine eigene Systemdatei besitzen, benötigen eine entsprechende physische Datenbank des Datenbanktyps ADA.

Natural-Objekte, die mit dem Datenbanktyp ADA kompiliert wurden, können in einer Umgebung ausgeführt werden, in der die entsprechende Datenbank als Datenbank vom Typ ADA2 definiert ist.

Datendefinitionsmodule (DDMs)

Damit Natural auf eine Datenbank-Datei zugreifen kann, ist eine logische Definition der physischen Datenbank-Datei erforderlich. Eine solche logische Dateidefinition wird DDM (Datendefinitionsmodul) genannt.

Dieser Abschnitt behandelt folgende Themen:

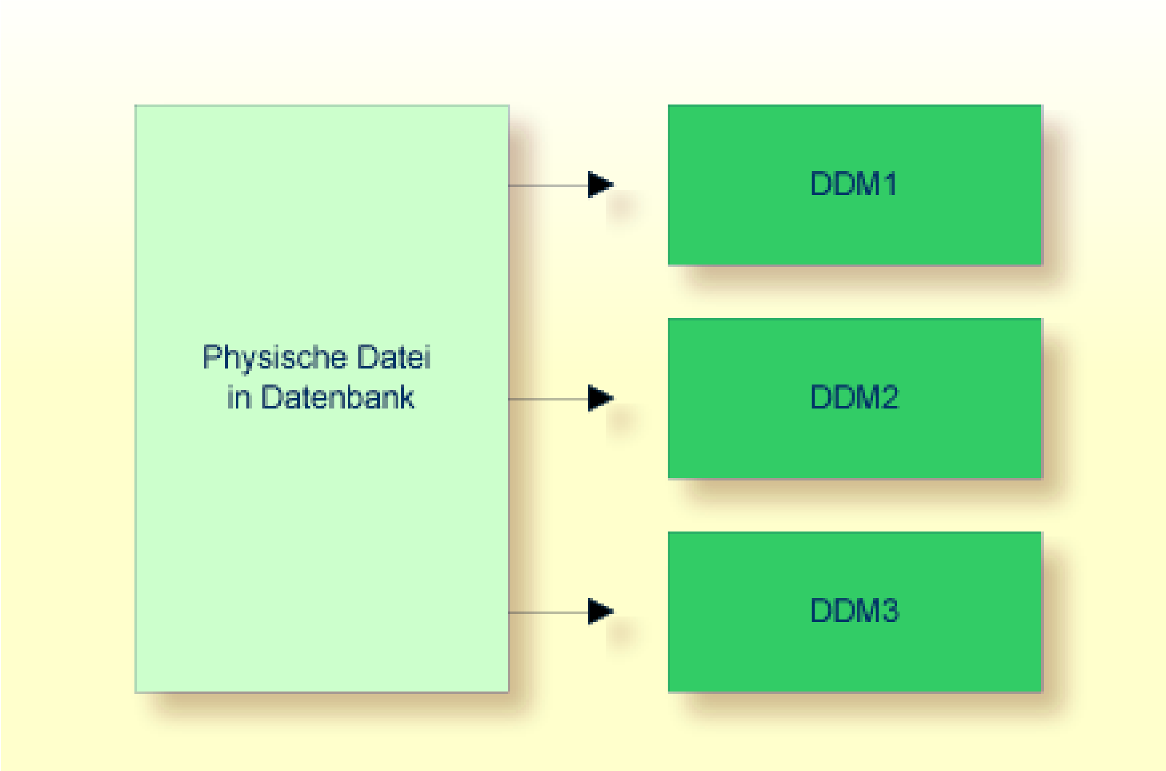
- Datendefinitionsmodule benutzen
- DDMs verwalten

- DDMs auflisten/anzeigen

Datendefinitionsmodule benutzen

Das DDM enthält Informationen über die einzelnen Felder der Datei — Informationen, die bei der Verwendung dieser Felder in einem Natural-Programm relevant sind. Ein DDM stellt eine logische Sicht (View) auf eine physische Datenbank-Datei dar.

Für jede physische Datei einer Datenbank können ein oder mehrere DDMs definiert werden. Und für jedes DDM können ein oder mehrere Datensichten definiert werden (siehe *View-Definition* in der `DEFINE DATA`-Statement-Dokumentation).



DDMs werden vom Natural-Administrator mit Predict definiert (oder, falls Predict nicht vorhanden ist, mit der entsprechenden Natural-Funktion zum Verwalten von DDMs).

DDMs verwalten

Ein DDM enthält die datenbankinternen Feldnamen der Datenbankfelder sowie ihre “externen” Feld-Langnamen (d.h. die in einem Natural-Programm verwendeten Feldnamen). Außerdem sind im DDM Format und Länge der Felder definiert, sowie weitere Angaben, die verwendet werden, wenn ein Feld in einem `DISPLAY`- oder `WRITE`-Statement benutzt wird (Spaltenüberschriften, Editiermasken usw.).

Informationen zu den in einem DDM definierten Feldattributen entnehmen Sie dem Abschnitt *Using the DDM Editor Windows* unter *DDM Editor* in der Natural *Editors*-Dokumentation.

DDMs auflisten/anzeigen

Wenn Sie den Namen einer benötigten DDM Nicht kennen, können Sie sich mit dem Systemkommando `LIST VIEW` eine Liste aller in der aktuellen Library vorhandenen Views (d.h. DDMs) anzeigen lassen. Aus der Liste können Sie dann eine DDM zur Anzeige auswählen.

LIST VIEW	Zeigt eine Liste aller Views (DDMs).
LIST VIEW *`view-name`*	Wenn Sie einen einzelnen View-Namen angeben, wird die angegebene View angezeigt. Für den *`view-name`* können Sie Stern-Notation (*) oder einen bestimmten Bereich von Views angeben (zum Beispiel: A*).

Datenbank-Arrays

Adabas unterstützt Array-Strukturen innerhalb der Datenbank in Form von *multiplen Feldern* und *Periodengruppen*.

Dieser Abschnitt behandelt folgende Themen:

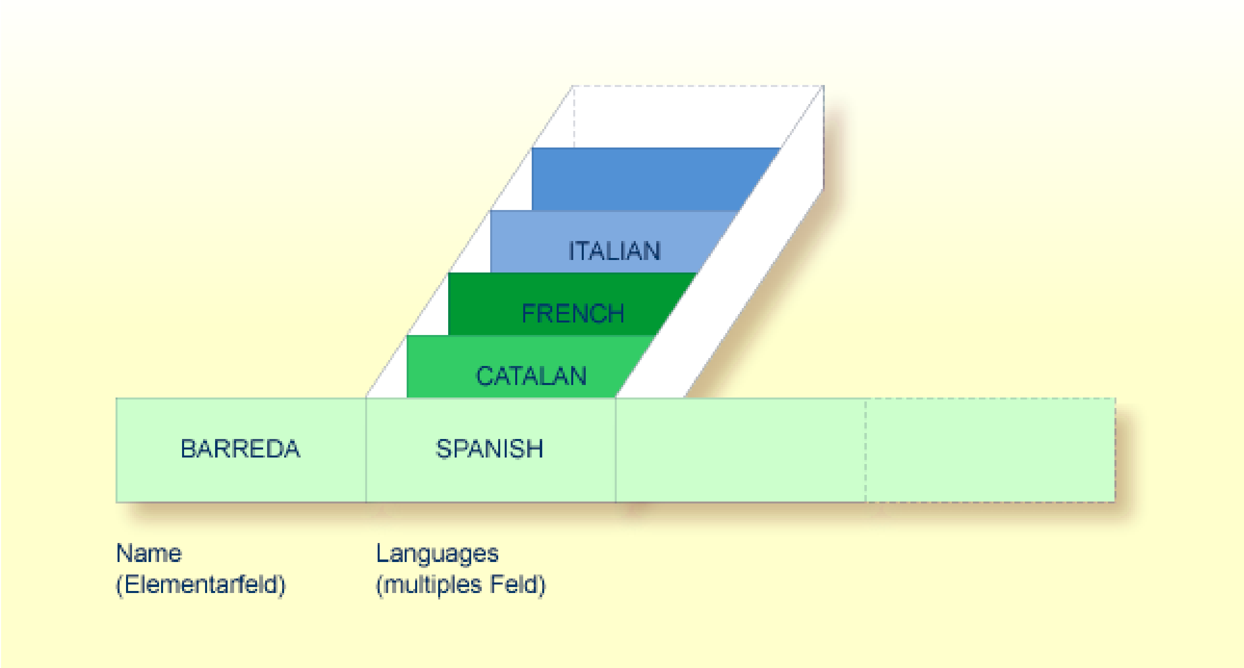
- Multiple Felder
- Periodengruppen
- Multiple Felder und Periodengruppen referenzieren
- Multiple Felder innerhalb von Periodengruppen
- Multiple Felder innerhalb von Periodengruppen referenzieren

- Internen Zähler eines Datenbank-Arrays referenzieren

Multiple Felder

Ein *multiples Feld* ist ein Feld, das innerhalb eines Datensatzes mehr als einen Wert (bis zu 191) haben kann.

Beispiel:



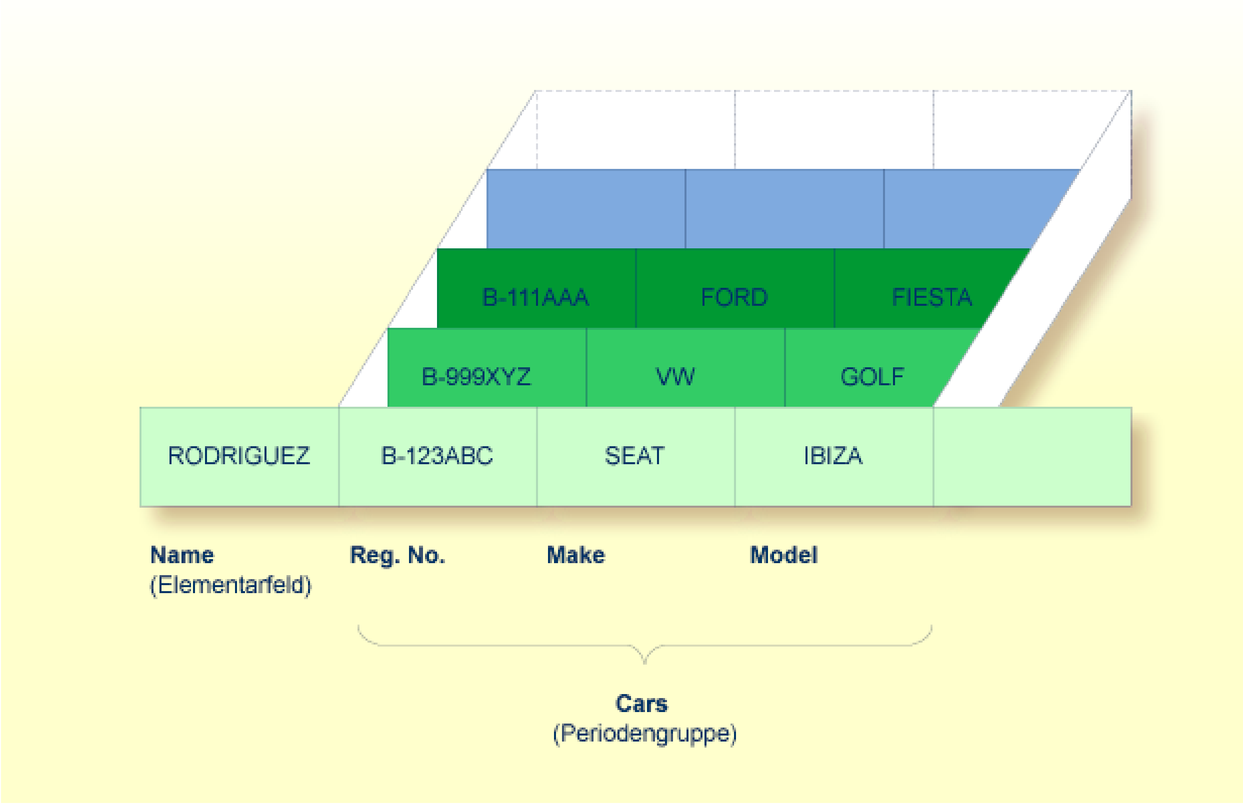
Angenommen, obige Abbildung zeigt einen Datensatz aus einer Personaldatei: das erste Feld (`Name`) ist ein Elementarfeld, das nur einen Wert enthalten kann, nämlich den Namen der Person; das zweite Feld (`Languages`) enthält die Sprachen, die die Person spricht, und ist ein multiples Feld, da eine Person mehrere Sprachen sprechen kann.

Periodengruppen

Eine *Periodengruppe* ist eine Gruppe von Feldern (wobei es sich um Elementarfelder und/oder multiple Felder handeln kann), die innerhalb eines Datensatzes mehr als eine Ausprägung (bis zu 191) haben kann.

Bei multiplen Feldern werden die verschiedenen Werte eines Feldes auch als *Ausprägungen* bezeichnet, d.h. mit der Anzahl der Ausprägungen ist die Anzahl der Werte, die das Feld enthält, gemeint, und eine bestimmte Ausprägung bezeichnet einen bestimmten Wert. Analog dazu ist bei einer Periodengruppe mit Ausprägung eine Gruppe von Werten gemeint.

Beispiel:



Angenommen, obige Abbildung zeigt einen Datensatz aus einer Fahrzeugdatei: das erste Feld (`Name`) ist ein Elementarfeld, das den Namen einer Person enthält; `Cars` ist eine Periodengruppe, die die Fahrzeuge dieser Person enthält. Die Periodengruppe besteht aus drei Feldern, die für jedes Fahrzeug das KFZ-Kennzeichen (`Reg. No.`), die Marke (`Make`) und das Modell (`Model`) enthalten. Jede Ausprägung von `Cars` enthält jeweils die Werte für ein Fahrzeug.

Multiple Felder und Periodengruppen referenzieren

Um eine oder mehrere Ausprägungen eines multiplen Feldes oder einer Periodengruppe zu referenzieren, geben Sie hinter dem Feldnamen eine *Index-Notation* an.

Beispiele:

Die folgenden Beispiele verwenden das multiple Feld `LANGUAGES` und die Periodengruppe `CARS` aus den obigen Abbildung.

Die verschiedenen Werte des multiplen Feldes LANGUAGES können wie folgt referenziert werden:

LANGUAGES (1)	Referenziert den ersten Wert (SPANISH).
LANGUAGES (X)	Der Inhalt der Variablen X bestimmt den zu referenzierenden Wert.
LANGUAGES (1:3)	Referenziert die ersten drei Werte (SPANISH, CATALAN und FRENCH).
LANGUAGES (6:10)	Referenziert den sechsten bis zehnten Wert.
LANGUAGES (X:Y)	Die Inhalte der Variablen X und Y bestimmen die zu referenzierenden Werte.

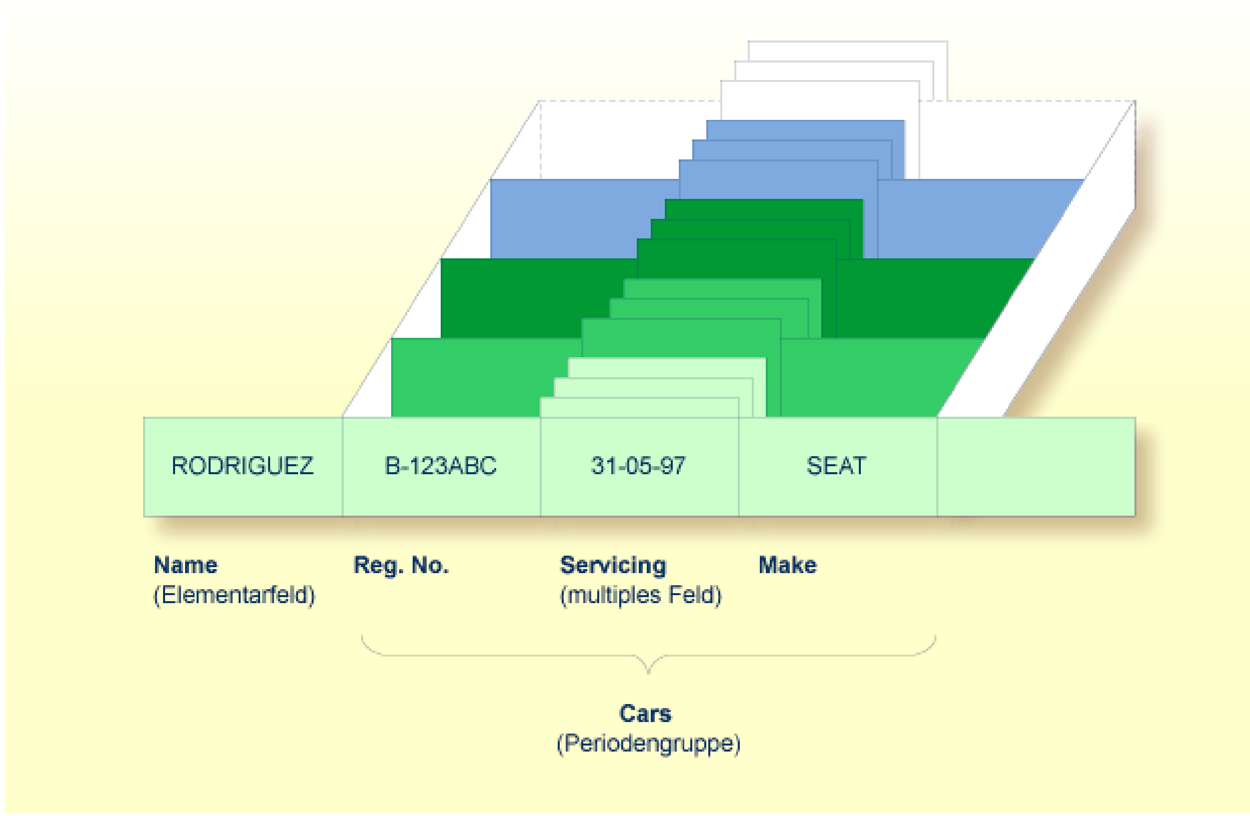
Die verschiedenen Ausprägungen der Periodengruppe CARS können in der gleichen Weise referenziert werden:

CARS (1)	Referenziert die erste Ausprägung (B-123ABC/SEAT/IBIZA).
CARS (X)	Der Inhalt der Variablen X bestimmt die zu referenzierende Ausprägung.
CARS (1:2)	Referenziert die ersten beiden Ausprägungen (B-123ABC/SEAT/IBIZA und B-999XYZ/VW/GOLF).
CARS (4:7)	Referenziert die vierte bis siebte Ausprägung.
CARS (X:Y)	Die Inhalte der Variablen X und Y bestimmen die zu referenzierenden Ausprägungen.

Multiple Felder innerhalb von Periodengruppen

Ein Adabas-Array kann bis zu zwei Dimensionen haben: ein multiples Feld innerhalb einer Periodengruppe.

Beispiel:



Angenommen, obige Abbildung zeigt einen Datensatz aus einer Fahrzeugdatei: das erste Feld (`Name`) ist ein Elementarfeld, das den Namen einer Person enthält; `Cars` ist eine Periodengruppe, die die Fahrzeuge dieser Person enthält. Die Periodengruppe besteht aus drei Feldern, die für jedes Fahrzeug das KFZ-Kennzeichen (`Reg. No.`), die Inspektionstermine (`Servicing`) und die Marke (`Make`) enthalten. Innerhalb der Periodengruppe `Cars` ist `Servicing` ein multiples Feld, das die verschiedenen Inspektionstermine jedes Autos enthält.

Multiple Felder innerhalb von Periodengruppen referenzieren

Um eine oder mehrere Ausprägungen eines multiplen Feldes innerhalb einer Periodengruppe zu referenzieren, geben Sie eine „zweidimensionale" Index-Notation hinter dem Feldnamen an.

Beispiele:

Die folgenden Beispiele verwenden das multiple Feld `SERVICING` und die Periodengruppe `CARS` aus der obigen Abbildung. Die verschiedenen Werte des multiplen Feldes können wie folgt referenziert werden:

`SERVICING (1,1)`	Referenziert den ersten Wert von `SERVICING` in der ersten Ausprägung von `CARS` (`31-05-97`).
`SERVICING (1:5,1)`	Referenziert jeweils den ersten Wert von `SERVICING` in den ersten fünf Ausprägungen von `CARS`.
`SERVICING (1:5,1:10)`	Referenziert jeweils die ersten zehn Werte von `SERVICING` in den ersten fünf Ausprägungen von `CARS`.

Internen Zähler eines Datenbank-Arrays referenzieren

Es ist manchmal erforderlich, ein multiples Feld oder eine Periodengruppe zu referenzieren, ohne die Anzahl der Werte bzw. Ausprägungen eines Datensatzes zu kennen. Adabas zählt intern die Anzahl der Werte eines multiplen Feldes und die Anzahl der Ausprägungen einer Periodengruppe. Dieser interne Zähler kann mit einem `READ`-Statement abgelesen werden, indem man unmittelbar vor dem Feldnamen `C*` angibt:

Die Anzahl wird jeweils in Format/Länge N3 zurückgegeben. Weitere Informationen entnehmen Sie dem Abschnitt ***Internen Zähler eines Datenbank-Arrays referenzieren*** in der *Statements*-Dokumentation.

Beispiele:

`C*LANGUAGES`	Liefert die Anzahl der Werte des multiplen Feldes `LANGUAGES`.
`C*CARS`	Liefert die Anzahl der Ausprägungen der Periodengruppe `CARS`.
`C*SERVICING(1)`	Liefert die Anzahl der Werte des multiplen Feldes `SERVICING` in der ersten Ausprägung einer Periodengruppe (ausgehend von der Annahme, dass `SERVICING` ein multiples Feld innerhalb einer Periodengruppe ist).

Datenbank-View definieren

Um Datenbankfelder in einem Natural-Programm verwenden zu können, müssen Sie sie in einer sogenannten View (Datenbanksicht) angeben.

In dem View geben Sie Folgendes an: den Namen des Datendefinitionsmoduls (siehe ***Datendefinitionsmodule (DDMs)***), aus dem die Felder stammen, und die **Namen der Datenbankfelder** selbst (d.h. ihre Langnamen, nicht ihre datenbankinternen Kurznamen).

Ein View kann ein komplettes DDM umfassen oder einen Ausschnitt daraus. Die Reihenfolge der Felder in der View braucht nicht mit der Reihenfolge der Felder im zugrundeliegenden DDM übereinzustimmen.

Wie im Abschnitt ***Statements für Datenbankzugriffe*** noch gezeigt wird, wird der View-Name in den Statements `READ`, `FIND`, `HISTOGRAM` verwendet, um zu bestimmen, auf welche Datenbank zugegriffen werden soll.

Weitere Informationen bezüglich der vollständigen Syntax der View-Definition oder über die Definition/Redifinition einer Gruppe von Feldern siehe *View-Definition* in der Beschreibung des `DEFINE DATA`-Statements in der *Statements*-Dokumentation.

Sie haben folgende Möglichkeiten, um eine Datenbank-View zu definieren:

- **Innerhalb des Programms**
  Sie können eine Datenbank-View innerhalb des Programms, d.h. direkt im `DEFINE DATA`-Statement des Programms definieren.
- **Außerhalb des Programms**
  Sie können eine Datenbank-View außerhalb des Programms, d.h. in einem separaten Programmierobjekt definieren: entweder in einer Local Data Area (LDA) oder in einer Global Data Area (GDA), wobei das `DEFINE DATA`-Statement dann diese Data Area referenziert.

Um eine Datenbank-View innerhalb des Programms zu definieren

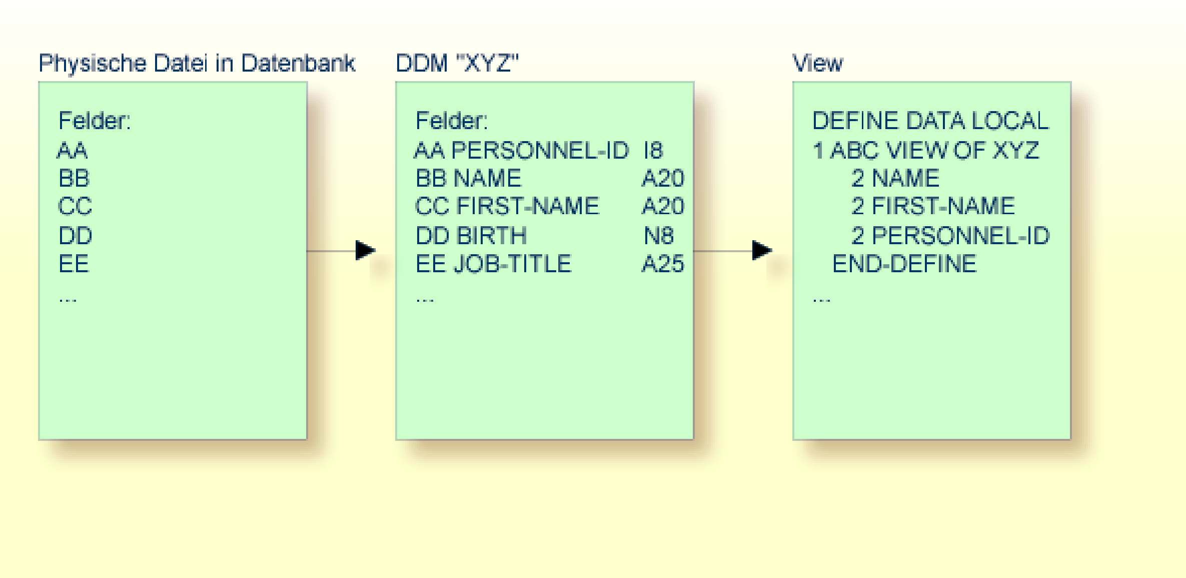
1 Auf Level 1 geben Sie den View-Namen wie folgt an:

```
1 view-name VIEW OF ddm-name
```

wobei *view-name* der von Ihnen gewählte Name für den View ist, und *ddm-name* der Name des DDMs, aus dem die im View angegebenen Felder stammen.

2 Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

In der folgenden Abbildung hat die View den Namen ABC und umfasst die Felder NAME, FIRST-NAME und PERSONNEL-ID aus dem DDM XYZ.



Format und Länge eines Datenbankfeldes brauchen in der View nicht angegeben zu werden, da sie bereits im zugrundeliegenden DDM definiert sind.

Beispiel-Programm:

In diesem Beispiel lautet der View-Name VIEWEMP, der DDM-Name ist EMPLOYEES und die Namen der aus dem DDM stammenden Felder lauten NAME, FIRST-NAME und PERSONNEL-ID.

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

Um eine Datenbank-View außerhalb des Programms zu definieren

1 Im Programm selbst geben Sie an:

```
DEFINE DATA LOCAL
       USING <data-area-name>
END-DEFINE
  ...
```

wobei *data-area-name* der von Ihnen gewählte Name für die Local or Global Data Area ist, zum Beispiel `LDA39`.

2 In der im Programm referenzierten Data Area geben Sie Folgendes an:

1. Auf Level 1 in der Spalte `Name` den Namen, den Sie für die View gewählt haben, und in der Spalte `Miscellaneous` den Namen des DDM, aus dem die in der View angegebenen Felder stammen.
2. Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

   Beispiel-Data-Area `LDA39`:

   In diesem Beispiel lautet der View-Name `VIEWEMP`, der DDM-Name ist `EMPLOYEES` und die Namen der aus dem DDM stammenden Felder lauten `PERSONNEL-ID`, `FIRST-NAME` und `NAME`.

```
I T L  Name                             F Length     Miscellaneous

All -- -------------------------------- - ---------- ------------------------->
  V  1 VIEWEMP                                       EMPLOYEES

     2 PERSONNEL-ID                     A          8

     2 FIRST-NAME                       A         20

     2 NAME                             A         20

     1 #VARI-A                          A         20

     1 #VARI-B                          N        3.2

     1 #VARI-C                          I          4
```

Anmerkungen zu Datenbanken des Typs ADA2

Bei Datenbanken des Typs ADA2 (Angabe erfolgt in der Tabelle DBMS Assignments in der Configuration Utility, siehe *Database Management System Assignments* in der *Configuration Utility* Dokumentation), gilt Folgendes:

- Wenn Felder mit großen alphanumerischen Variablen (LA) oder großen Objekten (LOB) verwendet werden sollen (Adabas LA/LB-Option), können diese Felder in der View-Definition sowohl mit festen Format/Längen-Werten, zum Beispiel A20 or U20, als auch mit dynamischen Format/Länge-Werten, zum Beispiel (A)DYNAMIC or U(DYNAMIC), angegeben werden.
- Außerdem können Längen-Indikator-Felder L@... in Views angegeben werden, wenn sie sich auf LA- bzw. LB-Felder beziehen.

Statements für Datenbankzugriffe

Um Daten von einer Datenbank zu lesen, stehen folgende Statements zur Verfügung:

READ	Mit diesem Statement können Sie eine Reihe von Datensätzen in einer bestimmten Reihenfolge von der Datenbank lesen.
FIND	Mit diesem Statement können Sie von einer Datenbank diejenigen Datensätze lesen, die ein bestimmtes Suchkriterium erfüllen.
HISTOGRAM	Mit diesem Statement können Sie nur die Werte eines einzelnen Datenbankfeldes lesen oder herausfinden, wieviele Datensätze ein bestimmtes Suchkriterium erfüllen.

Das READ-Statement

Folgende Themen werden behandelt:

- Verwendung des READ-Statements
- Syntax-Grundform des READ-Statements
- Beispiel für READ-Statement:
- Anzahl der zu lesenden Datensätze begrenzen
- STARTING- und ENDING-Klausel beim READ-Statement
- WHERE-Klausel beim READ-Statement

- Weiteres Beispiel für READ-Statement

Verwendung des READ-Statements

Das READ-Statement dient dazu, Datensätze von einer Datenbank zu lesen. Die Datensätze können von der Datenbank gelesen werden:

- in der Reihenfolge, in der sie physisch auf der Datenbank gespeichert sind (READ IN PHYSICAL SEQUENCE) oder
- in der Reihenfolge der Adabas-internen Satznummern (READ BY ISN) oder
- in logischer Reihenfolge der Werte eines Deskriptorfeldes (READ IN LOGICAL SEQUENCE).

In diesem Handbuch wird lediglich READ IN LOGICAL SEQUENCE behandelt, da dies die am häufigsten verwendete Form des READ-Statements ist.

Informationen zu den anderen beiden Möglichkeiten finden Sie unter der Beschreibung des READ-Statements in der *Statements*-Dokumentation.

Syntax-Grundform des READ-Statements

Die Grundform des READ-Statements ist:

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

oder kürzer:

```
READ view LOGICAL BY descriptor
```

- dabei ist

view	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt ***Datenbank-View definieren*** beschrieben).
descriptor	der Name eines in diesem View definierten Datenbankfeldes. Die Werte dieses Feldes bestimmen die Reihenfolge, in der die Datensätze von der Datenbank gelesen werden.

Wenn Sie einen Deskriptor angeben, erübrigt sich die Angabe des **Schlüsselwortes** `LOGICAL`:

```
READ view BY descriptor
```

Wenn Sie keinen Deskriptor angeben, werden die Datensätze in der Reihenfolge der Werte des im DDM als Standard-Deskriptor (unter "Default Sequence") definierten Feldes gelesen. Wenn Sie keinen Deskriptor angeben, müssen Sie allerdings das Schlüsselwort `LOGICAL` angeben:

```
READ view LOGICAL
```

Beispiel für READ-Statement:

```
** Example 'READX01': READ
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 JOB-TITLE
END-DEFINE
*
READ (6) MYVIEW BY NAME
  DISPLAY NAME PERSONNEL-ID JOB-TITLE
END-READ
END
```

Ausgabe des Programms `READX01`:

Das `READ`-Statement im obigen Beispiel liest Datensätze von der Mitarbeiter-Datei `EMPLOYEES` in alphabetischer Reihenfolge der (im Feld `NAME` enthaltenen) Nachnamen.

Das obige Programm erzeugt folgende Ausgabe, wobei die Informationen zu jedem Mitarbeiter in alphabetischer Reihenfolge der Nachnamen angezeigt werden:

```
Page      1                                                04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                        ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             2001100
```

Falls Sie die Mitarbeiterdaten in der Reihenfolge der (im Feld BIRTH enthaltenen) Geburtsdaten lesen und ausgeben möchten, wäre dazu folgendes READ-Statement geeignet:

```
READ MYVIEW BY BIRTH
```

Sie können nur ein Feld angeben, das im zugrundeliegenden **DDM** als *Deskriptor* definiert ist (es kann auch ein Subdeskriptor, Superdeskriptor, Hyperdeskriptor, phonetischer Deskriptor oder Nicht-Deskriptor sein).

Anzahl der zu lesenden Datensätze begrenzen

Wie im Beispielprogramm auf der vorigen Seite gezeigt, können Sie die Anzahl der Datensätze, die gelesen werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort READ in Klammern eine Zahl angeben:

```
READ (6) MYVIEW BY NAME
```

In diesem Beispiel würde das READ-Statement maximal 6 Datensätze lesen.

Ohne diese Limit-Notation würde das obige READ-Statement *sämtliche* Datensätze von der EMPLOYEES-Datei in der Reihenfolge der Nachnamen von A bis Z lesen.

STARTING- und ENDING-Klausel beim READ-Statement

Mit dem READ-Statement können Sie das Suchkriterium für die zu lesenden Datensätze durch einen bestimmten *Wert* eines Deskriptorfeldes weiter einschränken. Mit der Option EQUAL TO/STARTING FROM in einer BY bzw. WITH-Klausel können Sie festlegen, ab welchem Wert die Datensätze gelesen werden sollen. Mit der Option THRU/ENDING AT können Sie darüber hinaus bestimmen, bis zu welchem Wert gelesen werden soll.

Wünschen Sie beispielsweise eine Liste aller Mitarbeiter in der Reihenfolge der Tätigkeitsbezeichnungen (JOB-TITLE) von TRAINEE bis Z, würden Sie eines der folgenden Statements verwenden:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
READ MYVIEW WITH JOB-TITLE STARTING from 'TRAINEE'
READ MYVIEW BY JOB-TITLE = 'TRAINEE'
READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE'
```

Bitte beachten Sie, dass der Wert hinter dem Gleichheitszeichen (=) bzw. der STARTING FROM-Option in Apostrophen (') stehen muss. Bei einem numerischen Wert ist diese **Text-Notation** nicht erforderlich.

Es ist nicht möglich, die Optionen BY und WITH gleichzeitig zu verwenden; es ist jeweils nur eine von beiden gestattet.

Durch Angabe einer THRU bzw. ENDING AT-Klausel können Sie darüber hinaus festlegen, bis zu welchem Punkt Datensätze gelesen werden sollen.

Um nur Datensätze mit der Tätigkeitsbezeichnung TRAINEE zu lesen, müssten Sie folgendes angeben:

```
READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE' THRU 'TRAINEE'
READ MYVIEW WITH JOB-TITLE EQUAL TO 'TRAINEE'
                           ENDING AT 'TRAINEE'
```

Um alle Datensätze mit Tätigkeitsbezeichnungen, die mit A oder B anfangen, zu lesen, müssten Sie folgendes angeben:

```
READ MYVIEW BY JOB-TITLE = 'A' THRU 'C'
READ MYVIEW WITH JOB-TITLE STARTING from 'A' ENDING AT 'C'
```

Die Werte werden gelesen bis einschließlich des Wertes, der nach THRU/ENDING AT spezifiziert wird. In den beiden obigen Beispielen werden alle Datensätze mit Tätigkeitsbezeichnungen, die mit A oder B anfangen, gelesen; gäbe es eine Tätigkeitsbezeichnung C, würde diese auch gelesen werden, aber nicht der nächsthöhere Wert CA.

WHERE-Klausel beim READ-Statement

Mit einer WHERE-Klausel können Sie ein zusätzliches Suchkriterium angeben.

Zum Beispiel, wenn Sie nur die Datensätze derjenigen Mitarbeiter mit Tätigkeitsbezeichnung TRAINEE, die in US-Währung (USD) bezahlt werden, lesen wollen, dann geben Sie Folgendes an:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
            WHERE CURR-CODE = 'USD'
```

Die WHERE-Klausel kann auch zusammen mit einer BY-Klausel verwendet werden, zum Beispiel:

```
READ MYVIEW BY NAME
            WHERE SALARY = 20000
```

Die WHERE-Klausel unterscheidet sich in zwei Punkten von einer BY/WITH-Klausel:

- Das in der WHERE-Klausel angegebene Feld muss kein Deskriptor sein.
- In der WHERE-Klausel wird eine logische Bedingung angegeben.

Folgende logische Operatoren können in einer WHERE-Klausel verwendet werden:

EQUAL	EQ	=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS THAN OR EQUAL TO	LE	<=
GREATER THAN	GT	>
GREATER THAN OR EQUAL TO	GE	>=

Das folgende Programm veranschaulicht die Verwendung der Klauseln STARTING FROM, ENDING AT und WHERE:

```
** Example 'READX02': READ  (with STARTING, ENDING and WHERE clause)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME     (1:2)
    3 CURR-CODE
    3 SALARY
    3 BONUS    (1:1)
END-DEFINE
*
READ (3) MYVIEW WITH  JOB-TITLE
STARTING FROM 'TRAINEE' ENDING AT 'TRAINEE'
                WHERE CURR-CODE (*) = 'USD'
  DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
  SKIP 1
END-READ
END
```

Ausgabe des Programms READX02:

```
          NAME                                INCOME
         CURRENT
        POSITION              CURRENCY    ANNUAL     BONUS
                                CODE      SALARY
-------------------------     -------- ---------- ----------

SENKO                         USD          23000          0
TRAINEE                       USD          21800          0

BANGART                       USD          25000          0
TRAINEE                       USD          23000          0

LINCOLN                       USD          24000          0
TRAINEE                       USD          22000          0
```

Weiteres Beispiel für READ-Statement

Siehe folgendes Beispiel-Programm:

- *READX03 - READ-Statement*

Das FIND-Statement

Folgende Themen werden behandelt:

- Verwendung des FIND-Statements
- Syntax-Grundform des FIND Statements
- Anzahl der zu verarbeitenden Datensätze begrenzen
- WHERE-Klausel beim FIND-Statement
- Beispiel für FIND-Statement mit WHERE-Klausel:
- IF NO RECORDS FOUND-Bedingung
- Weitere Beispiele zum FIND-Statement

Verwendung des FIND-Statements

Das `FIND`-Statement dient dazu, Datensätze von einer Datenbank zu lesen, die ein bestimmtes Suchkriterium erfüllen.

Syntax-Grundform des FIND Statements

Die Grundform des `FIND`-Statements ist:

```
FIND RECORDS IN view WITH field = value
```

oder kürzer:

```
FIND view WITH field = value
```

- dabei ist

view	der Name eines im `DEFINE DATA`-Statement definierten Views (wie im Abschnitt ***Datenbank-View definieren*** beschrieben).
field	der Name eines in diesem View definierten Datenbankfeldes.

Sie können nur ein *field* angeben, das im zugrundeliegenden **DDM** als *Deskriptor* definiert ist (es kann auch ein Subdeskriptor, Superdeskriptor, Hyperdeskriptor, phonetischer Deskriptor oder ein Nicht-Deskriptor sein).

Die vollständige Syntax entnehmen Sie der `FIND`-Statement-Dokumentation.

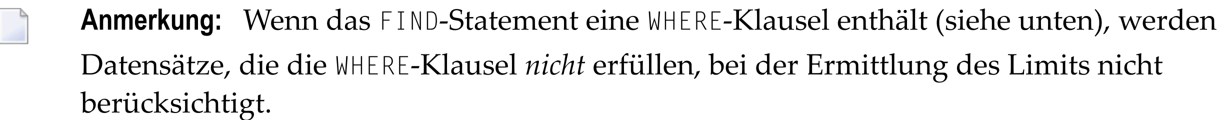
Anzahl der zu verarbeitenden Datensätze begrenzen

Ähnlich wie beim READ-Statement (siehe **oben**) können Sie die Anzahl der Datensätze, die verarbeitet werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort `FIND` in Klammern eine Zahl angeben:

```
FIND (6) RECORDS IN MYVIEW WITH NAME = 'CLEGG'
```

In diesem Beispiel würde das `FIND`-Statement maximal 6 Datensätze verarbeiten.

Ohne diese Limit-Notation würden alle Datensätze, die das Suchkriterium erfüllen, verarbeitet werden.



Anmerkung: Wenn das `FIND`-Statement eine `WHERE`-Klausel enthält (siehe unten), werden Datensätze, die die `WHERE`-Klausel *nicht* erfüllen, bei der Ermittlung des Limits nicht berücksichtigt.

WHERE-Klausel beim FIND-Statement

Mit der `WHERE`-Klausel des `FIND`-Statements können Sie ein zusätzliches Selektionskriterium angeben, das ausgewertet wird, *nachdem* ein (über die `WITH`-Klausel ausgewählter) Datensatz gelesen wurde und *bevor* der ausgewählte Datensatz weiterverarbeitet wird.

Beispiel für FIND-Statement mit WHERE-Klausel:

```
** Example 'FINDX01': FIND (with WHERE)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
            WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
  DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END
```

Anmerkung: Wie Sie sehen, werden in diesem Beispiel nur die Datensätze, die die Kriterien der `WITH`-Klausel *und* der `WHERE`-Klausel erfüllen, im `DISPLAY`-Statement verarbeitet.

Ausgabe des Programms `FINDX01`:

```
        CITY                     CURRENT          PERSONNEL         NAME
                                POSITION             ID
-------------------- ------------------------- --------- --------------------

PARIS                INGENIEUR COMMERCIAL      50007300  CAHN
PARIS                INGENIEUR COMMERCIAL      50006500  MAZUY
PARIS                INGENIEUR COMMERCIAL      50004700  FAURIE
PARIS                INGENIEUR COMMERCIAL      50004400  VALLY
PARIS                INGENIEUR COMMERCIAL      50002800  BRETON
PARIS                INGENIEUR COMMERCIAL      50001000  GIGLEUX
PARIS                INGENIEUR COMMERCIAL      50000400  KORAB-BRZOZOWSKI
```

IF NO RECORDS FOUND-Bedingung

Falls keine Datensätze gefunden werden, die die in der WITH- und WHERE-Klausel angegebenen Suchkriterien erfüllen, werden die innerhalb der FIND-Verarbeitungsschleife angegebenen Statements nicht ausgeführt (für das Beispiel auf der vorigen Seite hieße dies, dass das DISPLAY-Statement nicht ausgeführt würde und folglich keine Mitarbeiterdaten angezeigt würden).

Das FIND-Statement bietet jedoch auch eine IF NO RECORDS FOUND-Klausel, in der Sie eine Verarbeitung angeben können, die ausgeführt werden soll für den Fall, dass kein Datensatz die Suchkriterien erfüllt.

Beispiel für FIND-Statement mit IF NO RECORDS FOUND-Bedingung:

```
** Example 'FINDX02': FIND (with IF NO RECORDS FOUND)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKSMITH'
  IF NO RECORDS FOUND
    WRITE 'NO PERSON FOUND.'
  END-NOREC
  DISPLAY NAME FIRST-NAME
END-FIND
END
```

Das obige Programm wählt alle Datensätze aus, in denen das Feld NAME den Wert BLACKSMITH enthält. Von jedem ausgewählten Datensatz werden der Name (NAME) und der Vorname (FIRST-NAME) angezeigt. Falls in der Datei kein Datensatz mit NAME = 'BLACKSMITH' gefunden wird, wird das in der IF NO RECORDS FOUND-Klausel angegebene WRITE-Statement ausgeführt:

Ausgabe des Programms FINDX02:

```
Page      1                                                 04-11-11  14:15:54

        NAME              FIRST-NAME
-------------------- --------------------

NO PERSON FOUND.
```

Weitere Beispiele zum FIND-Statement

Siehe die folgenden Beispiel-Programme:

- *FINDX07 - FIND (mit mehreren Klauseln)*
- *FINDX08 - FIND (mit LIMIT)*
- *FINDX09 - FIND (unter Verwendung von *NUMBER, *COUNTER, *ISN)*
- *FINDX10 - FIND (in Kombination mit READ)*
- *FINDX11 - FIND NUMBER (mit *NUMBER)*

Das HISTOGRAM-Statement

Folgende Themen werden behandelt:

- Verwendung des HISTOGRAM-Statements
- Syntax-Grundform des HISTOGRAM-Statements
- Anzahl der zu lesenden Werte begrenzen
- STARTING- und ENDING-Klausel beim HISTOGRAM-STATEMENT
- WHERE-Klausel beim HISTOGRAM-Statement
- Beispiel für HISTOGRAM-Statement

Verwendung des HISTOGRAM-Statements

Das `HISTOGRAM`-Statement dient dazu, entweder die Werte eines einzelnen Datenbankfeldes zu lesen oder herauszufinden, wieviele Datensätze ein bestimmtes Suchkriterium erfüllen.

Das `HISTOGRAM`-Statement kann auf keine anderen Datenbankfelder zugreifen als auf das im `HISTOGRAM`-Statement angegebene Feld.

Syntax-Grundform des HISTOGRAM-Statements

Die Grundform des `HISTOGRAM`-Statements ist:

```
HISTOGRAM VALUE IN view FOR field
```

oder kürzer:

```
HISTOGRAM view FOR field
```

- dabei ist

view	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt ***Datenbank-View definieren*** beschrieben).
field	der Name des in diesem View definierten Datenbankfeldes.

Die vollständige Syntax entnehmen Sie der HISTOGRAM-Statement-Dokumentation.

Anzahl der zu lesenden Werte begrenzen

Ähnlich wie beim READ-Statement (siehe **oben**) können Sie die Anzahl der Werte, die gelesen werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort HISTOGRAM in Klammern eine Zahl angeben:

```
HISTOGRAM (6) MYVIEW FOR NAME
```

In diesem Beispiel würden nur die ersten 6 Werte des Feldes NAME gelesen.

Ohne diese Limit-Notation würden alle Werte gelesen.

STARTING- und ENDING-Klausel beim HISTOGRAM-STATEMENT

Wie das READ-Statement (siehe **oben**) bietet auch das HISTOGRAM-Statement eine STARTING FROM-Klausel- und eine ENDING AT bzw. THRU-Klausel, mit denen Sie den Bereich der zu lesenden Werte durch Angabe eines Startwertes und eines Endwertes eingrenzen können.

Beispiele:

```
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD'
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD' ENDING AT 'LANIER'
HISTOGRAM MYVIEW FOR NAME from 'BLOOM' THRU 'ROESER'
```

WHERE-Klausel beim HISTOGRAM-Statement

Das `HISTOGRAM`-Statement bietet außerdem eine `WHERE`-Klausel, in der Sie ein zusätzliches Selektionskriterium angeben können, das ausgewertet wird, *nachdem* ein Wert gelesen wurde und *bevor* der Wert weiterverarbeitet wird. Das in der `WHERE`-Klausel angegebene Feld muss dasselbe sein wie das in der Hauptklausel des `HISTOGRAM`-Statements angegebene.

Beispiel für HISTOGRAM-Statement

```
** Example 'HISTOX01': HISTOGRAM
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
LIMIT 8
HISTOGRAM MYVIEW CITY STARTING FROM 'M'
  DISPLAY NOTITLE CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
END-HISTOGRAM
END
```

In diesem Programm werden mit dem `HISTOGRAM`-Statement außerdem die Systemvariablen `*NUMBER` und `*COUNTER` ausgewertet und mit dem `DISPLAY`-Statement ausgegeben. `*NUMBER` enthält die Anzahl der Datensätze, in denen der zuletzt gelesene Wert vorkommt; `*COUNTER` enthält die Gesamtanzahl der bisher gelesenen Werte.

Ausgabe des Programms `HISTOX01`:

```
        CITY          NUMBER OF     CNT
                       PERSONS
-------------------- ----------- -----------

MADISON                        3           1
MADRID                        41           2
MAILLY LE CAMP                 1           3
MAMERS                         1           4
MANSFIELD                      4           5
MARSEILLE                      2           6
MATLOCK                        1           7
MELBOURNE                      2           8
```

Multi-Fetch-Klausel

Dieser Abschnitt behandelt die *Multi-Fetch*-Datensatz-Retrieval-Funktionalität für Adabas-Datenbanken.

Die Multi-Fetch-Funktionalität wird nur bei Datenbanken des Typs `ADA` unterstützt, die in der Tabelle DBMS Assignments in der Configuration Utility angegeben werden; siehe *Database Management System Assignments* in der *Configuration Utility*-Dokumentation. Bei Datenbanken des Typs ADA2 wird die Multi-Fetch-Funktionalität nicht unterstützt.

Folgende Themen werden behandelt:

- Sinn und Zweck der Multi-Fetch-Funktion
- Statements mit Multi-Fetch-Verarbeitung
- Bemerkungen zur Multi-Fetch-Benutzung

Sinn und Zweck der Multi-Fetch-Funktion

Im Standardmodus liest Natural mit einem einzigen Datenbank-Aufruf nicht mehrere Datensätze ein, sondern stets nur ein Datensatz pro Fetch-Modus. Diese Art von Betrieb ist solide und stabil, kann aber einige Zeit in Anspruch nehmen, wenn eine große Anzahl von Datenbank-Sätzen verarbeitet wird. Um die Verarbeitungszeit dieser Programme zu verbessern, können Sie die Multi-Fetch-Verarbeitung nutzen.

Standardmäßig verwendet Natural den Single-Fetch-Modus, um Daten aus einer Adabas-Datenbank abzufragen. Der Abfrage-Modus kann mit dem Natural-Profilparameter `MFSET` konfiguriert werden.

Mit den Werten `ON` (Multi-Fetch) und `OFF` (Single-Fetch) wird das Standard-Verhalten festgelegt. Wenn `MFSET` auf `NEVER` gesetzt wird, verwendet Natural immer den Single-Fetch-Modus und ignoriert Einstellungen auf Statement-Ebene.

Der Standard-Verarbeitungsmodus kann außerdem auf Statement-Ebene geändert werden.

Statements mit Multi-Fetch-Verarbeitung

Die Multi-Fetch-Verarbeitung wird bei folgenden Statements unterstützt, die keine Datenbank verändernden Zugriffe ausführen:

- `FIND`
- `READ`
- `HISTOGRAM`

Informationen zur Syntax finden Sie in der Beschreibung der `MULTI-FETCH`-Klausel bei den Statements `FIND`, `READ` und `HISTOGRAM`.

Bemerkungen zur Multi-Fetch-Benutzung

Falls geschachtelte Datenbankabfrageschleifen, die sich auf dieselbe Adabas-Datei beziehen, in einer der inneren Schleifen `UPDATE`-Statements enthalten, macht Natural bei der Verarbeitung der äußeren Schleifen mit den aktualisierten Werten weiter. Im Multi-Fetch-Modus bedeutet dies, dass eine äußere logische `READ`-Schleife repositioniert werden muss, wenn eine innere Datenbankabfrageschleife den Wert aktualisiert, der zur Steuerung der Reihenfolge in der äußeren Schleife benutzt wird. Wenn dieser Versuch einen Konflikt für den aktuellen Deskriptor verursacht, wird ein Fehler zurückgegeben. Um dies zu vermeiden, empfiehlt es sich, dass Sie die Multi-Fetch-Funktion in der äußeren Schleife ausschalten.

Generell ist festzustellen, dass die Multi-Fetch-Verarbeitung die Leistung bei Zugriffen auf eine Adabas-Datenbank erhöht. In manchen Fällen kann es dennoch vorteilhaft sein, den Single-Fetch-Modus zur Leistungssteigerung zu verwenden, insbesondere wenn auch Datenbank verändernde Zugriffe ausgeführt werden.

Datenbank-Verarbeitungsschleifen

Dieser Abschnitt erörtert Verarbeitungsschleifen, die zum Abarbeiten von Daten erforderlich sind, welche von einer Datenbank als Ergebnis eines `FIND`-, `READ`- oder `HISTOGRAM`-Statements ausgewählt wurden.

Folgende Themen werden behandelt:

- Erstellung von Datenbank-Verarbeitungsschleifen
- Hierarchien von Verarbeitungsschleifen
- Beispiel für geschachtelte FIND-Schleifen, die dieselbe Datei aufrufen
- Weitere Beispiele für geschachtelte READ- und FIND-Statements

Erstellung von Datenbank-Verarbeitungsschleifen

Natural initiiert automatisch die Schleifen, die zur Verarbeitung von Daten erforderlich sind, die mit einem `FIND`-, `READ`- oder `HISTOGRAM`-Statement von einer Datenbank ausgewählt wurden.

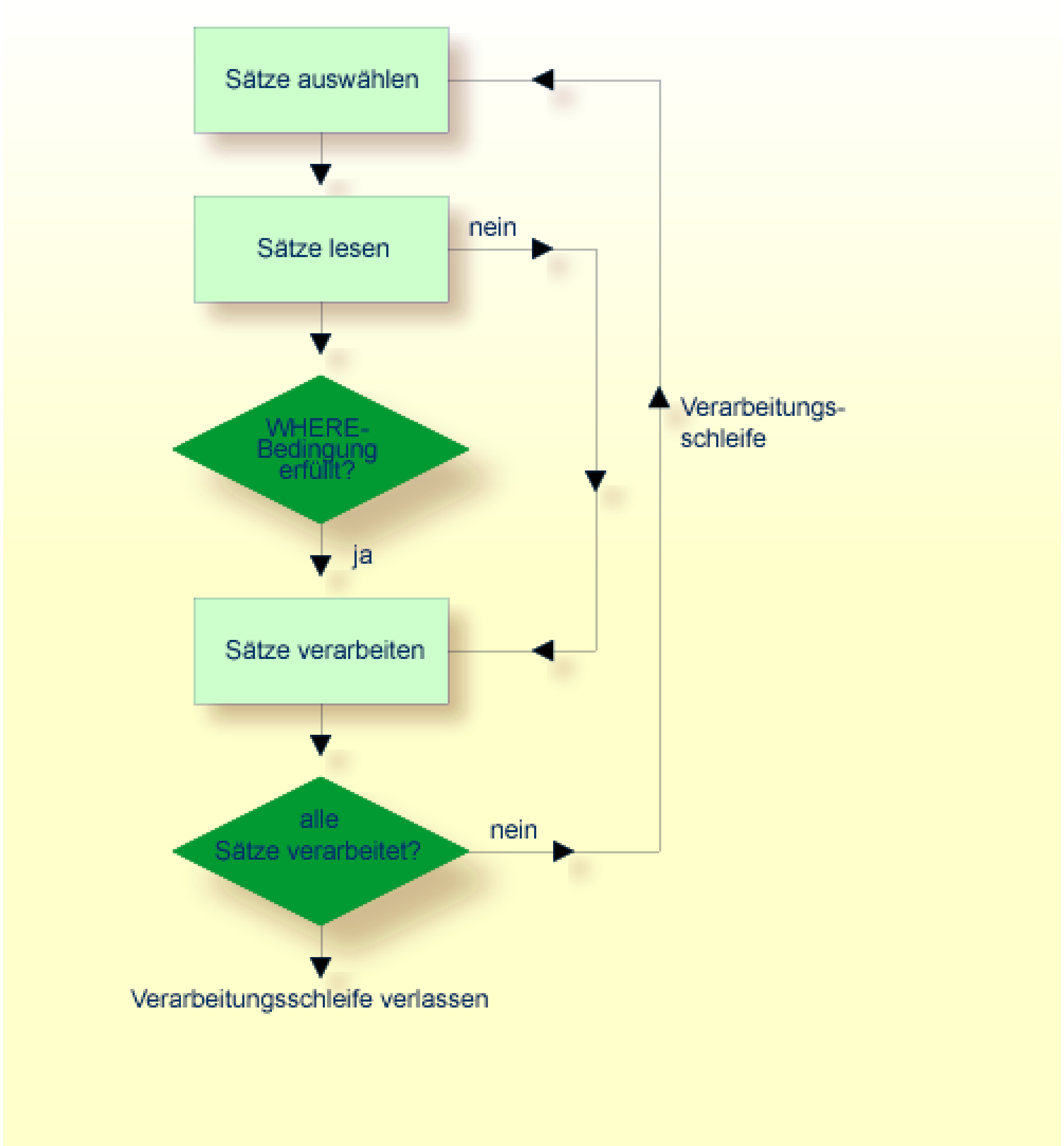
Beispiel:

Die obige `FIND`-Schleife wählt von der Datei `EMPLOYEES` alle Datensätze aus, in denen das Feld `NAME` den Wert `ADKINSON` enthält, und verarbeitet die ausgewählten Datensätze. Im Beispiel besteht die Verarbeitung in der Anzeige bestimmter Feldwerte aus jedem der ausgewählten Datensätze.

```
** Example 'FINDX03': FIND
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH NAME = 'ADKINSON'
  DISPLAY NAME FIRST-NAME CITY
END-FIND
END
```

Wenn obiges `FIND`-Statement zusätzlich zu der `WITH`-Klausel noch eine `WHERE`-Klausel enthielte, würden nur diejenigen der ausgewählten Datensätze verarbeitet, die die `WITH`- *und* die `WHERE`-Bedingung erfüllen.

Das folgende Diagramm zeigt den logischen Ablauf einer Datenbank-Verarbeitungsschleife:


Sätze auswählen
Sätze lesen
nein
WHERE-Bedingung erfüllt?
ja
Verarbeitungs-schleife
Sätze verarbeiten
alle Sätze verarbeitet?
nein
Verarbeitungsschleife verlassen

Hierarchien von Verarbeitungsschleifen

Die Verwendung mehrerer FIND- bzw. READ-Statements führt zu einer Hierarchie ineinander geschachtelter Schleifen, wie das folgende Beispiel zeigt:

Beispiel für Verarbeitungsschleifen-Hierarchie:

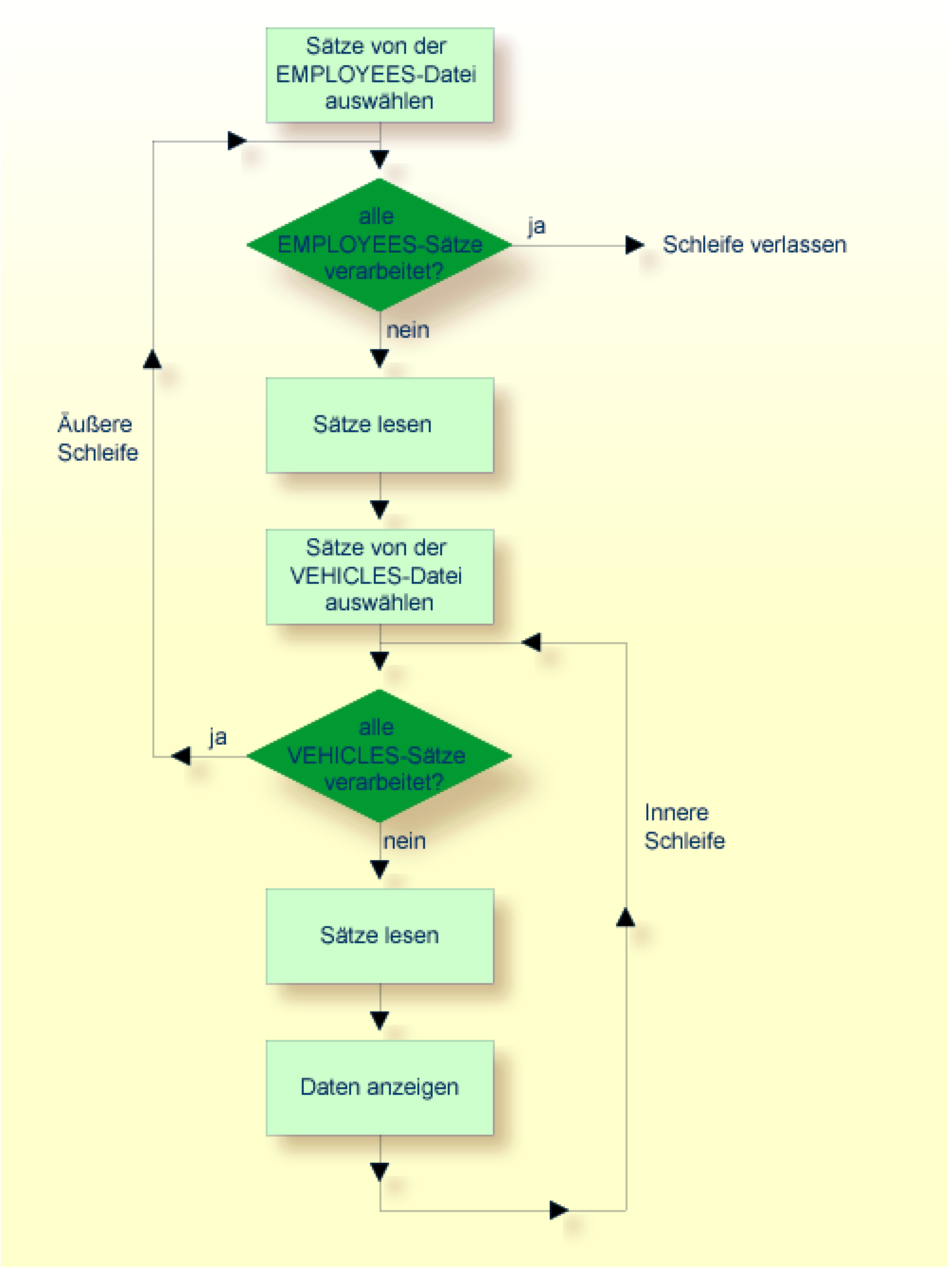
```
** Example 'FINDX04': FIND  (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 AUTOVIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
  2 MODEL
END-DEFINE
*
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'
  VEH. FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    DISPLAY NAME MAKE MODEL
  END-FIND
END-FIND
END
```

Im obigen Programm werden zunächst alle Datensätze mit Namen ADKINSON von der Datei EMPLOYEES ausgewählt. Dann wird jeder dieser Datensätze (jede Person) wie folgt verarbeitet:

1. Mit dem zweiten FIND-Statement werden für alle von der Datei EMPLOYEES gelesenen Personen die dazugehörigen Fahrzeuge (VEHICLES) gesucht, und zwar unter Verwendung der Personalnummern (PERSONNEL-ID) aus den mit dem ersten FIND-Statement von der EMPLOYEES-Datei ausgewählten Datensätzen.
2. Dann werden mit DISPLAY folgende Werte angezeigt: der NAME jeder gefundenen Person (diese Informationen werden von der EMPLOYEES-Datei gelesen) und Marke und Modell (MAKE und MODEL) des dazugehörigen Fahrzeugs (diese Informationen kommen von der VEHICLES-Datei).

Das zweite FIND-Statement initiiert innerhalb der äußeren FIND-Schleife des ersten FIND-Statements eine innere Schleife, wie das folgende Diagramm veranschaulicht.

Das folgende Diagramm zeigt den logischen Ablauf in der Datenbank-Verarbeitungsschleifen-Hierarchie in dem obigen Beispiel-Programm:


Sätze von der EMPLOYEES-Datei auswählen
alle EMPLOYEES-Sätze verarbeitet?
ja
Schleife verlassen
nein
Sätze lesen
Äußere Schleife
Sätze von der VEHICLES-Datei auswählen
ja
alle VEHICLES-Sätze verarbeitet?
nein
Innere Schleife
Sätze lesen
Daten anzeigen

Beispiel für geschachtelte FIND-Schleifen, die dieselbe Datei aufrufen

Es ist auch möglich, eine Verarbeitungsschleifen-Hierarchie aufzubauen, in der zwei ineinander verschachtelte Schleifen auf dieselbe Datei zugreifen, wie das folgende Beispiel zeigt.

```
** Example 'FINDX05': FIND (two FIND statements on same file nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #NAME (A40)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED
  'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
*
FIND PERSONVIEW WITH NAME = 'JONES'
                WHERE FIRST-NAME = 'LAUREL'
  COMPRESS NAME FIRST-NAME INTO #NAME
  /*
  FIND PERSONVIEW WITH CITY = CITY
    DISPLAY NAME FIRST-NAME CITY
  END-FIND
END-FIND
END
```

Im obigen Programm werden zunächst in der Datei `EMPLOYEES` alle Personen mit Namen `JONES` und Vornamen `LAUREL` gesucht. Dann werden zu jeder gefundenen Person alle Personen, die in derselben Stadt wohnen, in der `EMPLOYEES`-Datei gesucht, und es wird eine Liste dieser Personen erzeugt. Alle mit dem `DISPLAY`-Statement ausgegebenen Feldwerte werden mit dem zweiten `FIND`-Statement gelesen.

Ausgabe des Programms `FINDX05`:

```
PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

        NAME                 FIRST-NAME              CITY
-------------------- -------------------- --------------------

JENSON               MARTHA               BALTIMORE
LAWLER               EDDIE                BALTIMORE
FORREST              CLARA                BALTIMORE
ALEXANDER            GIL                  BALTIMORE
NEEDHAM              SUNNY                BALTIMORE
```

```
ZINN                CARLOS              BALTIMORE
JONES               LAUREL              BALTIMORE
```

Weitere Beispiele für geschachtelte READ- und FIND-Statements

Siehe die folgenden Beispiel-Programme:

- ***READX04 - READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)***
- ***LIMITX01 - LIMIT-Statement (für READ- und FIND-Schleifenverarbeitung)***

Datenänderungen - Transaktionsverarbeitung

Dieser Abschnitt beschreibt, wie Natural Datenbankänderungsoperationen mittels Transaktionen durchführt.

Folgende Themen werden behandelt:

- Logische Transaktionen
- Datensatz-Kontrolle während einer Transaktion (Hold-Logik)
- Transaktion abbrechen
- Transaktion neu starten
- Beispiel für Verwendung von Transaktionsdaten beim Neustarten einer Transaktion

Logische Transaktionen

Natural führt Veränderungen auf der Datenbank auf der Grundlage von Transaktionen aus, d.h. alle Veränderungszugriffe werden in logische Transaktionseinheiten gegliedert. Eine Transaktion ist die kleinste (von Ihnen definierte) Verarbeitungseinheit, die vollständig ausgeführt werden muss, um die logische Konsistenz der gespeicherten Daten zu gewährleisten.

Eine logische Transaktion kann aus einem oder mehreren datenverändernden Statements (`DELETE`, `STORE`, `UPDATE`) bestehen und auf eine oder mehrere Dateien zugreifen. Eine logische Transaktion kann sich auch über mehrere Natural-Programme erstrecken.

Eine logische Transaktion beginnt, sobald ein Datensatz in den Hold-Status gestellt wird. Dies erfolgt durch Natural automatisch, wenn ein Satz zwecks Änderung gelesen wird, wenn also z.B. in einer `FIND`-Schleife ein `UPDATE`- oder `DELETE`-Statement steht.

Das Ende einer logischen Transaktion wird im Programm durch ein `END TRANSACTION`-Statement bestimmt. Dieses Statement gewährleistet, dass alle durch die Transaktion bewirkten Änderungen erfolgreich durchgeführt werden, und gibt anschließend alle während der Transaktion gehaltenen Datensätze wieder frei.

Beispiel:

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
   DELETE
   END TRANSACTION
END-FIND
END
```

Jeder gefundene Satz würde hier in den Hold-Status gestellt, gelöscht und anschließend, wenn das `END TRANSACTION`-Statement ausgeführt wird, aus dem Hold-Status wieder freigegeben.

Anmerkung: Mit dem Natural-Profilparameter `ETEOP` kann der Natural-Administrator festlegen, ob Natural am Ende jedes Programms ein `END TRANSACTION`-Statement generieren soll. Einzelheiten hierzu sagt Ihnen Ihr Natural-Administrator.

Beispiel für ein STORE-Statement:

In dem folgenden Beispiel-Programm werden neue Datensätze der `EMPLOYEES`-Datei hinzugefügt.

Vorsicht: Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```
** Example 'STOREX01': STORE  (Add new records to EMPLOYEES file)
*
** CAUTION: Executing this example will modify the database records!
***********************************************************************
DEFINE DATA LOCAL
1 EMPLOYEE-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID(A8)
  2 NAME        (A20)
  2 FIRST-NAME  (A20)
  2 MIDDLE-I    (A1)
  2 SALARY      (P9/2)
  2 MAR-STAT    (A1)
  2 BIRTH       (D)
  2 CITY        (A20)
  2 COUNTRY     (A3)
*
1 #PERSONNEL-ID (A8)
1 #NAME         (A20)
1 #FIRST-NAME   (A20)
1 #INITIAL      (A1)
1 #MAR-STAT     (A1)
1 #SALARY       (N9)
```

```
1 #BIRTH        (A8)
1 #CITY         (A20)
1 #COUNTRY      (A3)
1 #CONF         (A1)   INIT <'Y'>
END-DEFINE
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
        'PERSONNEL-ID : ' #PERSONNEL-ID    //
        'NAME         : ' #NAME  /
        'FIRST-NAME   : ' #FIRST-NAME
  /*********************************************************************
  /*  validate entered data
  /*********************************************************************
  IF #PERSONNEL-ID = 'END' OR #NAME = 'END'
    STOP
  END-IF
  IF #NAME = ' '
    REINPUT WITH TEXT 'ENTER A LAST-NAME'
            MARK 2 AND SOUND ALARM
  END-IF
  IF #FIRST-NAME = ' '
    REINPUT WITH TEXT 'ENTER A FIRST-NAME'
            MARK 3 AND SOUND ALARM
  END-IF
  /*********************************************************************
  /*  ensure person is not already on file
  /*********************************************************************
  FIP2. FIND NUMBER EMPLOYEE-VIEW WITH PERSONNEL-ID =  #PERSONNEL-ID
  /*
  IF *NUMBER (FIP2.) > 0
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
             MARK 1 AND SOUND ALARM
  END-IF
  /*********************************************************************
  /*  get further information
  /*********************************************************************
  INPUT
    'ENTER EMPLOYEE DATA'                                ////
    'PERSONNEL-ID              :' #PERSONNEL-ID (AD=IO) /
    'NAME                      :' #NAME         (AD=IO) /
    'FIRST-NAME                :' #FIRST-NAME   (AD=IO) ///
    'INITIAL                   :' #INITIAL              /
    'ANNUAL SALARY             :' #SALARY               /
    'MARITAL STATUS            :' #MAR-STAT             /
    'DATE OF BIRTH (YYYYMMDD) :' #BIRTH                /
    'CITY                      :' #CITY                 /
    'COUNTRY (3 CHARS)         :' #COUNTRY              //
    'ADD THIS RECORD (Y/N)     :' #CONF         (AD=M)
  /*********************************************************************
  /*  ENSURE REQUIRED FIELDS CONTAIN VALID DATA
  /*********************************************************************
```

```
  IF #SALARY < 10000
    REINPUT TEXT 'ENTER A PROPER ANNUAL SALARY' MARK 2
  END-IF
  IF NOT (#MAR-STAT = 'S'  OR = 'M' OR = 'D' OR = 'W')
    REINPUT TEXT 'ENTER VALID MARITAL STATUS  S=SINGLE ' -
                 'M=MARRIED D=DIVORCED W=WIDOWED' MARK 3
  END-IF
  IF NOT(#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
    REINPUT TEXT 'ENTER CORRECT DATE' MARK 4
  END-IF
  IF #CITY  = ' '
    REINPUT TEXT 'ENTER A CITY NAME' MARK 5
  END-IF
  IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 6
  END-IF
  IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 7
  END-IF
  IF #CONF = 'N'
    ESCAPE TOP
  END-IF
  /*********************************************************************
  /*  add the record with STORE
  /*********************************************************************
  MOVE #PERSONNEL-ID TO EMPLOYEE-VIEW.PERSONNEL-ID
  MOVE #NAME         TO EMPLOYEE-VIEW.NAME
  MOVE #FIRST-NAME   TO EMPLOYEE-VIEW.FIRST-NAME
  MOVE #INITIAL      TO EMPLOYEE-VIEW.MIDDLE-I
  MOVE #SALARY       TO EMPLOYEE-VIEW.SALARY (1)
  MOVE #MAR-STAT     TO EMPLOYEE-VIEW.MAR-STAT
  MOVE EDITED #BIRTH TO EMPLOYEE-VIEW.BIRTH (EM=YYYYMMDD)
  MOVE #CITY         TO EMPLOYEE-VIEW.CITY
  MOVE #COUNTRY      TO EMPLOYEE-VIEW.COUNTRY
  /*
  STP3. STORE RECORD IN FILE EMPLOYEE-VIEW
  /*
  /*********************************************************************
  /*  mark end of logical transaction
  /*********************************************************************
  END OF TRANSACTION
  RESET INITIAL #CONF
END-REPEAT
END
```

Ausgabe des Programms `STOREX01`:

```
ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

NAME         :
FIRST NAME   :
```

Datensatz-Kontrolle während einer Transaktion (Hold-Logik)

Wird Natural zusammen mit Adabas eingesetzt, so wird jeder Datensatz, der verändert werden soll, solange in den Hold-Status gestellt, bis die Transaktion entweder durch ein `END TRANSACTION`- oder `BACKOUT TRANSACTION`-Statement beendet oder aufgrund einer Zeitüberschreitung abgebrochen wird.

Solange ein Datensatz für einen Benutzer im Hold-Status steht, haben andere Benutzer keine Möglichkeit, diesen Datensatz zu ändern. Ein Benutzer, der dies tun will, gelangt in den Wartestatus (Wait) und erhält die Kontrolle über den gewünschten Satz erst, wenn der erste Benutzer seine Transaktion beendet/abgebrochen hat.

Um zu verhindern, dass ein Benutzer im Wartestatus verbleibt, ist es möglich den Session-Parameter `WH` (Wait Hold) entsprechend zu setzen (siehe *Parameter-Referenz*-Dokumentation).

Beim Programmieren sollten Sie folgendes bezüglich der Hold-Logik bedenken:

- Die Zeit, für die ein Datensatz höchstens in den Hold-Status gestellt werden kann, wird von Adabas durch das *Transaction Time Limit* (Transaktionszeitbegrenzung; Adabas `TT`-Parameter) begrenzt. Wird diese Zeit überschritten, erhält man eine entsprechende Fehlermeldung, und Veränderungen, die nach dem letzten `END TRANSACTION`-Statement erfolgten, werden rückgängig gemacht.
- Die Anzahl der ISNs im Hold-Status und mögliche Transaktionszeitüberschreitungen ergeben sich aus der Größe einer Transaktion, d.h. aus der Platzierung des `END TRANSACTION`-Statements. In diesem Zusammenhang sollten Sie die Nutzung von Restart-Möglichkeiten in Betracht ziehen. Falls die Mehrzahl der zu verarbeitenden Datensätze *nicht* verändert werden soll, empfiehlt es sich beispielsweise, ein `GET`-Statement zu verwenden, um das „Halten“ von Sätzen zu steuern. Damit spart man viele `END TRANSACTION`-Statements und verringert gleichzeitig die Zahl der in den Hold-Status gestellten ISNs. Bei Verarbeitung umfangreicher Dateien sollte bedacht werden, dass für ein GET-Statement ein zusätzlicher Adabas-Aufruf erforderlich ist. Ein Beispiel für die Verwendung eines `GET`-Statements sehen Sie im folgenden.

Beispiel für Hold-Logik:

Vorsicht: Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```
** Example 'GETX01': GET (put single record in hold with UPDATE stmt)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
*
RD. READ EMPLOY-VIEW BY NAME
  DISPLAY EMPLOY-VIEW
  IF SALARY (1) > 1500000
    /*
    GE. GET EMPLOY-VIEW *ISN (RD.)
    /*
    WRITE '=' (50) 'RECORD IN HOLD:' *ISN(RD.)
    COMPUTE SALARY (1) = SALARY (1) * 1.15
    UPDATE (GE.)
    END TRANSACTION
  END-IF
END-READ
END
```

Transaktion abbrechen

Innerhalb einer aktiven logischen Transaktion, d.h. bevor das `END TRANSACTION`-Statement ausgeführt wird, können Sie durch Verwendung eines `BACKOUT TRANSACTION`-Statements den Abbruch der Transaktion bewirken. Dadurch werden alle vorgenommenen Änderungen (einschließlich hinzugefügter und gelöschter Datensätze) rückgängig gemacht und die von der Transaktion gehaltenen Datensätze freigegeben.

Transaktion neu starten

Mit dem `END TRANSACTION`-Statement können Sie auch transaktionsbezogene Informationen speichern. Falls die Verarbeitung der Transaktion nicht ordnungsgemäß beendet werden kann, können Sie beim Neustarten (Restart) der Transaktion diese Informationen mit einem `GET TRANSACTION DATA`-Statement lesen, um festzustellen, an welchem Punkt die Verarbeitung fortgesetzt werden muss.

Beispiel für Verwendung von Transaktionsdaten beim Neustarten einer Transaktion

Im folgenden Beispielprogramm werden Daten der Dateien EMPLOYEES und VEHICLES verändert. Wenn das Programm nach einem Abbruch neu gestartet wird, werden Sie durch eine Restart-Prozedur darüber informiert, welcher Datensatz der Datei EMPLOYEES vor dem Abbruch zuletzt verarbeitet wurde, und können die Verarbeitung dann an dieser Stelle wiederaufnehmen. Es bestünde zusätzlich die Möglichkeit, Angaben über den zuletzt bearbeiteten Satz der VEHICLES-Datei in die Restart-Transaktionsmeldung einzufügen.

Vorsicht: Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```
** Example 'GETTRX01': GET TRANSACTION
*
** CAUTION: Executing this example will modify the database records!
***********************************************************************
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
  02 PERSONNEL-ID      (A8)
  02 NAME              (A20)
  02 FIRST-NAME        (A20)
  02 MIDDLE-I          (A1)
  02 CITY              (A20)
01 AUTO VIEW OF VEHICLES
  02 PERSONNEL-ID      (A8)
  02 MAKE              (A20)
  02 MODEL             (A20)
*
01 ET-DATA
  02 #APPL-ID          (A8) INIT <' '>
  02 #USER-ID          (A8)
  02 #PROGRAM          (A8)
  02 #DATE             (A10)
  02 #TIME             (A8)
  02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                     #DATE    #TIME    #PERSONNEL-NUMBER
*
IF  #APPL-ID NOT = 'NORMAL'     /* if last execution ended abnormally
AND #APPL-ID NOT = ' '
  INPUT (AD=OIL)
    //  20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
    /   20T '***********************************'
    /// 25T      'APPLICATION:' #APPL-ID
    /   32T             'USER:' #USER-ID
    /   29T          'PROGRAM:' #PROGRAM
    /   24T     'COMPLETED ON:' #DATE 'AT' #TIME
```

```
    /   20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
*
REPEAT
  /*
  INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
  /*
  IF #PERSONNEL-NUMBER = '99999999'
    ESCAPE BOTTOM
  END-IF
  /*
  FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
    IF NO RECORDS FOUND
      REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
    END-NOREC
    FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
      IF NO RECORDS FOUND
        WRITE 'PERSON DOES NOT OWN ANY CARS'
        ESCAPE BOTTOM
      END-NOREC
      IF *COUNTER (FIND2.) = 1     /* first pass through the loop
        INPUT (AD=M)
          /   20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
          /   20T '----------------------------'
          /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
          /   22T   'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
          /   22T   'CITY:' CITY
          /   22T   'MAKE:' MAKE
          /   21T  'MODEL:' MODEL
        UPDATE (FIND1.)            /* update the EMPLOYEES file
      ELSE                         /* subsequent passes through the loop
        INPUT NO ERASE (AD=M IP=OFF) //////// 28T MAKE / 28T MODEL
      END-IF
      /*
      UPDATE (FIND2.)              /* update the VEHICLES file
      /*
      MOVE *APPLIC-ID TO #APPL-ID
      MOVE *INIT-USER TO #USER-ID
      MOVE *PROGRAM   TO #PROGRAM
      MOVE *DAT4E     TO #DATE
      MOVE *TIME      TO #TIME
      /*
      END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                      #DATE    #TIME    #PERSONNEL-NUMBER
      /*
    END-FIND                  /* for VEHICLES    (FIND2.)
  END-FIND                    /* for EMPLOYEES   (FIND1.)
END-REPEAT                    /* for REPEAT
*
STOP                          /* Simulate abnormal transaction end
END TRANSACTION 'NORMAL  '
END
```

Datensätze mit ACCEPT/REJECT auswählen

Dieser Abschnitt behandelt die Statements ACCEPT und REJECT, die Sie zur Auswahl von Datensätzen anhand von Ihnen definierter logischer Auswahlkriterien verwenden können.

Folgende Themen werden behandelt:

- Mit ACCEPT und REJECT verwendbare Statements
- Beispiel für ein ACCEPT-Statement
- Logische Bedingungen in ACCEPT/REJECT-Statements
- Beispiel für ACCEPT-Statement mit AND-Operator
- Beispiel für REJECT-Statement mit OR-Operator
- Weitere Beispiele für ACCEPT- und REJECT-Statements

Mit ACCEPT und REJECT verwendbare Statements

Sie können ACCEPT und REJECT zusammen mit den folgenden Datenbankzugriffs-Statements verwenden:

- READ
- FIND
- HISTOGRAM

Beispiel für ein ACCEPT-Statement

```
** Example 'ACCEPX01': ACCEPT IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
ACCEPT IF SALARY (1) >= 40000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

Ausgabe des Programms ACCEPX01:

```
Page      1                                               04-11-11  11:11:11

       NAME                   CURRENT           ANNUAL
                              POSITION          SALARY
-------------------- ------------------------- ----------

ADKINSON             DBA                            46700
ADKINSON             MANAGER                        47000
ADKINSON             MANAGER                        47000
AFANASSIEV           DBA                            42800
ALEXANDER            DIRECTOR                       48000
ANDERSON             MANAGER                        50000
ATHERTON             ANALYST                        43000
ATHERTON             MANAGER                        40000
```

Logische Bedingungen in ACCEPT/REJECT-Statements

Mit einem ACCEPT- oder REJECT-Statement können Sie zusätzlich zu der WHERE- und WITH-Bedingung eines READ-Statements weitere logische Auswahlkriterien angeben.

Das ACCEPT- bzw. REJECT-Auswahlkriterium wird erst ausgewertet, *nachdem* die über das READ-Statement ausgewählten Datensätze gelesen worden sind.

Die folgenden logischen Operatoren können in einem ACCEPT- bzw. REJECT-Statement verwendet werden (weitere Einzelheiten siehe ***Logische Bedingungen***):

EQUAL	EQ	:=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS EQUAL	LE	<=
GREATER THAN	GT	>
GREATER EQUAL	GE	>=

Außerdem können Sie die Boole'schen Operatoren AND, OR und NOT zur Verknüpfung logischer Bedingungen in ACCEPT / REJECT-Statements einsetzen; mit Klammern können Sie die Bedingungen in logische Einheiten unterteilen, siehe folgende Beispiele.

Beispiel für ACCEPT-Statement mit AND-Operator

Das folgende Programm zeigt die Verwendung des Boole'schen Operators AND in einem ACCEPT-Statement:

```
** Example 'ACCEPX02': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
ACCEPT IF  SALARY (1) >= 40000
         AND SALARY (1) <= 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

Ausgabe des Programms ACCEPX02:

```
Page      1                                                04-12-14  12:22:01

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                            42800
ATHERTON             ANALYST                        43000
ATHERTON             MANAGER                        40000
```

Beispiel für REJECT-Statement mit OR-Operator

Das folgende Programm zeigt die Verwendung des Boole'schen Operators OR in einem REJECT-Statement. Das Programm erzeugt die gleiche Ausgabe wie das vorherige mit dem ACCEPT-Statement, da gleichzeitig die logischen Operatoren umgekehrt wurden:

```
** Example 'ACCEPX03': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
REJECT IF SALARY (1) < 40000
         OR SALARY (1) > 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

Ausgabe des Programms `ACCEPX03`:

```
Page      1                                                  04-12-14  12:26:27

        NAME                     CURRENT            ANNUAL
                                POSITION            SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                             42800
ATHERTON             ANALYST                         43000
ATHERTON             MANAGER                         40000
```

Weitere Beispiele für ACCEPT- und REJECT-Statements

Siehe die folgenden Beispiel-Programme:

- *ACCEPX04 - ACCEPT IF ... LESS THAN ...*
- *ACCEPX05 - ACCEPT IF ... AND ...*
- *ACCEPX06 - REJECT IF ... OR ...*

AT START/END OF DATA-Statements

Dieser Abschnitt erörtert die Vewendung der Statements `AT START OF DATA` und `AT END OF DATA`.

Folgende Themen werden behandelt:

- AT START OF DATA-Statement
- AT END OF DATA-Statement
- Beispiel für AT START OF DATA- und AT END OF DATA-Statement
- Weitere Beispiele für AT START OF DATA- und AT END OF DATA-Statement

AT START OF DATA-Statement

Mit dem Statement `AT START OF DATA` können Sie eine beliebige Verarbeitung angeben, die ausgeführt werden soll, nachdem der erste Datensatz einer Datenbank-Verarbeitungsschleife gelesen worden ist.

Das `AT START OF DATA`-Statement muss innerhalb der betreffenden Verarbeitungsschleife stehen.

Erzeugt das `AT START OF DATA`-Statement eine Ausgabe, so wird diese *vor der ersten Feldwert-Ausgabe* ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

AT END OF DATA-Statement

Mit dem Statement `AT END OF DATA` können Sie eine beliebige Verarbeitung angeben, die ausgeführt werden soll, nachdem alle Datensätze in einer Datenbank-Verarbeitungsschleife verarbeitet worden sind.

Das `AT END OF DATA`-Statement muss innerhalb der betreffenden Verarbeitungsschleife stehen.

Erzeugt das `AT END OF DATA`-Statement eine Ausgabe, so wird diese *nach der letzten Feldwert-Ausgabe* ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

Beispiel für AT START OF DATA- und AT END OF DATA-Statement

Das folgende Beispielprogramm veranschaulicht die Verwendung der Statements `AT START OF DATA` und `AT END OF DATA`.

Das `AT START OF DATA`-Statement enthält die Systemvariable `*TIME` zur Anzeige der Uhrzeit

Das `AT END OF DATA`-Statement enthält die Systemfunktion `OLD`, um den Namen der zuletzt ausgewählten Person anzuzeigen.

```
** Example 'ATSTAX01': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME  (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
  /*
 AT START OF DATA
    WRITE 'RUN TIME:' *TIME /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
*
AT END OF PAGE
  WRITE / 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END
```

Ausgabe des Programms `ATSTAX01`:

```
                  XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT

     NAME           CURRENT                 INCOME
                   POSITION
                                CURRENCY   ANNUAL     BONUS
                                  CODE     SALARY
--------------- --------------- -------- ---------- ----------

RUN TIME: 12:43:19.1

DUYVERMAN       PROGRAMMER      USD           34000          0
PRATT           SALES PERSON    USD           38000       9000
MARKUSH         TRAINEE         USD           22000          0
```

```
LAST PERSON SELECTED: MARKUSH

AVERAGE SALARY:      31333
```

Weitere Beispiele für AT START OF DATA- und AT END OF DATA-Statement

Siehe die folgenden Beispiel-Programme.

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE-Statement (in Kombination mit AT END OF DATA)*

Unicode-Daten

Natural ermöglicht es den Benutzern, auf Wide-Character-Fields mit dem Format W in einer Adabas-Datenbank zuzugreifen.

In diesem Abschnitt werden folgende Themen behandelt:

- Datendefinitionsmodul
- Zugriffskonfiguration
- Einschränkungen

Datendefinitionsmodul

Adabas Wide-Character-Fields (W) werden auf Natural-Format U (Unicode) abgebildet.

Die Längen-Definition für ein Natural-Feld vom Format U entspricht der Hälfte der Größe des Adabas-Feldes mit dem Format W. Ein Adabas Wide-Character-Field der Länge 200 wird zum Beispiel auf `(U100)` in Natural abgebildet.

Zugriffskonfiguration

Natural erhält Daten aus Adabas und sendet Daten zurück an Adabas mittels UTF-16 als gemeinsam benutzte Kodierung.

Diese Kodierung wird mit dem `OPRB`-Parameter angegeben und an Adabas mit der offenen Anforderung versandt. Sie wird für Wide-Character-Fields benutzt und gilt für die gesamte Adabas Benutzer-Session.

Einschränkungen

Wide-character-Fields (W) variabler Länge werden nicht unterstützt.

Sortierfolgen-Deskriptoren werden nicht unterstützt.

Weitere Informationen zu Adabas und Unicode-Unterstützung entnehmen Sie der spezifischen Adabas Produkt-Dokumentation.

30 Daten in einer SQL-Datenbank aufrufen

- Generating Natural DDMs 260
- Setting Natural Profile Parameters 260
- Natural DML Statements 261
- Natural SQL Statements 268
- Flexible SQL 276
- RDBMS-Specific Requirements and Restrictions 277
- Data-Type Conversion 280
- Date/Time Conversion 280
- Obtaining Diagnostic Information about Database Errors 282
- SQL Authorization 282

Dieses Kapitel beschreibt, wie Sie Natural mit SQL-Datenbanken unter Verwendung von Entire Access benutzen können. Informationen zur Installation und Konfiguration finden Sie in dem Teil *Natural and Entire Access* in der *Database Management System Interfaces*-Dokumentation und in der separaten *Entire Access*-Dokumentation.

Anmerkung: Die folgenden Abschnitte liegen nur in englischer Sprache vor.

Anmerkung: Im Prinzip gelten die in dem Dokument *Daten in einer Adabas-Datenbank aufrufen* enthaltenen Funktionen und Beispiele auch für die von Natural unterstützten SQL-Datenbanken. Falls es Unterschiede gibt, sind diese in den Dokumenten für die einzelnen Datenbankzugriffs-Statements beschrieben (siehe *Statements*-Dokumentation), in Abschnitten namens *Datenbank-spezifische Bemerkungen* oder in den Beschreibungen zu den einzelnen Natural-Parametern (siehe *Parameter-Referenz*-Dokumentation). Des Weiteren bietet Natural eine Reihe von speziellen Statements zum Zugriff auf SQL-Datenbanken.

Generating Natural DDMs

Entire Access is an application programming interface (API) that supports Natural SQL statements and most Natural DML statements.

Natural DML and SQL statements can be used in the same Natural program. At compilation, if a DML statement references a DDM for a data source defined in *NATCONF.CFG* with DBMS type "SQL", Natural translates the DML statement into an SQL statement.

Natural converts DML and SQL statements into calls to Entire Access. Entire Access converts the requests to the data formats and SQL dialect required by the target RDBMS and passes the requests to the database driver.

Setting Natural Profile Parameters

ETEOP Parameter

This parameter can be set only by Natural administrators.

The Natural profile parameter `ETEOP` controls transaction processing during a Natural session. It is required, for example, if a single logical transaction is to span two or more Natural programs. In this case, Natural must not issue an `END TRANSACTION` command (that is, not „commit“) at the termination of a Natural program.

If the ETEOP parameter is set to:

ON	Natural issues an END TRANSACTION statement (that is, automatically „commits“) at the end of a Natural program if the Natural session is not at ET status.
OFF	Natural does not issue an END TRANSACTION command (that is, does not „commit“) at the end of a Natural program. This setting thus enables a single logical transaction to span more than one Natural program. This is the default.

Anmerkung: The ETEOP parameter applies to Natural Version 6.1 and above. With previous Natural versions, the Natural profile parameter OPRB has to be used instead of ETEOP (ETEOP=ON corresponds to OPRB=OFF, ETEOP=OFF corresponds to ORPB=NOOPEN).

Natural DML Statements

The following table shows how Natural translates DML statements into SQL statements:

DML Statement	SQL Statement
BACKOUT TRANSACTION	ROLLBACK
DELETE	DELETE WHERE CURRENT OF *cursor-name*
END TRANSACTION	COMMIT
EQUAL ... OR	IN (...)
EQUAL ... THRU ...	BETWEEN ... AND ...
FIND ALL	SELECT
FIND NUMBER	SELECT COUNT (*)
HISTOGRAM	SELECT COUNT (*)
READ LOGICAL	SELECT ... ORDER BY
READ PHYSICAL	SELECT ... ORDER BY
SORTED BY ... [DESCENDING]	ORDER BY ... [DESCENDING]
STORE	INSERT
UPDATE	UPDATE WHERE CURRENT of *cursor-name*
WITH	WHERE

Anmerkung: Boolean and relational operators function the same way in DML and SQL statements.

Entire Access does not support the following DML statements and options:

- CIPHER

- COUPLED
- FIND FIRST, FIND UNIQUE, FIND ... RETAIN AS
- GET, GET SAME, GET TRANSACTION DATA, GET RECORD
- PASSWORD
- READ BY ISN
- STORE USING/GIVING NUMBER

BACKOUT TRANSACTION

Natural translates a BACKOUT TRANSACTION statement into an SQL ROLLBACK command. This statement reverses all database modifications made after the completion of the last recovery unit. A recovery unit may start at the beginning of a session or after the last END TRANSACTION (COMMIT) or BACKOUT TRANSACTION (ROLLBACK) statement.

Anmerkung: Because all cursors are closed when a logical unit of work ends, do not place a BACKOUT TRANSACTION statement within a database loop; place it outside the loop or after the outermost loop of nested loops.

DELETE

The DELETE statement deletes a row from a database table that has been read with a preceding FIND, READ, or SELECT statement. It corresponds to the SQL statement DELETE WHERE CURRENT OF *cursor-name*, which means that only the last row that was read can be deleted.

Example:

```
FIND EMPLOYEES WITH NAME = 'SMITH'
     AND FIRST_NAME = 'ROGER'
DELETE
```

Natural translates the Natural statements above into the following SQL statements and assigns a cursor name (for example, "CURSOR1"). The SELECT statement and the DELETE statement refer to the same cursor.

```
SELECT FROM EMPLOYEES
 WHERE NAME = 'SMITH' AND FIRST_NAME = 'ROGER'
DELETE FROM EMPLOYEES
 WHERE CURRENT OF CURSOR1
```

Natural translates a DELETE statement into an SQL DELETE statement the way it translates a FIND statement into an SQL SELECT statement. For details, see the FIND statement description **below**.

Anmerkung: You cannot delete a row read with a `FIND SORTED BY` or `READ LOGICAL` statement. For an explanation, see the `FIND` and `READ` statement descriptions below.

END TRANSACTION

Natural translates an `END TRANSACTION` statement into an SQL `COMMIT` command. The `END TRANSACTION` statement indicates the end of a logical transaction, commits all modifications to the database, and releases data locked during the transaction.

Anmerkungen:

1. Because all cursors are closed when a logical unit of work ends, do not place an `END TRANSACTION` statement within a database loop; place it outside the loop or after the outermost loop of nested loops.
2. The `END TRANSACTION` statement cannot be used to store transaction (ET) data when used with Entire Access.
3. Entire Access does not issue a `COMMIT` automatically when the Natural program terminates.

FIND

Natural translates a `FIND` statement into an SQL `SELECT` statement. The `SELECT` statement is executed by an `OPEN CURSOR` command followed by a `FETCH` command. The `FETCH` command is executed repeatedly until all records have been read or the program exits the `FIND` processing loop. A `CLOSE CURSOR` command ends the `SELECT` processing.

Example:

Natural statements:

```
FIND EMPLOYEES WITH NAME = 'BLACKMORE'
   AND AGE EQ 20 THRU 40
OBTAIN PERSONNEL_ID NAME AGE
```

Equivalent SQL statement:

```
SELECT PERSONNEL_ID, NAME, AGE
  FROM EMPLOYEES
    WHERE NAME = 'BLACKMORE'
      AND AGE BETWEEN 20 AND 40
```

You can use any table column (field) designated as a descriptor to construct search criteria.

Natural translates the WITH clause of a FIND statement into the WHERE clause of an SQL SELECT statement. Natural evaluates the WHERE clause of the FIND statement after the rows have been selected using the WITH clause. View fields may be used in a WITH clause only if they are designated as descriptors.

Natural translates a FIND NUMBER statement into an SQL SELECT statement containing a COUNT(*) clause. When you want to determine whether a record exists for a specific search condition, the FIND NUMBER statement provides better performance than the IF NO RECORDS FOUND clause.

Anmerkung: A row read with a FIND statement containing a SORTED BY clause cannot be updated or deleted. Natural translates the SORTED BY clause of a FIND statement into the ORDER BY clause of an SQL SELECT statement, which produces a read-only result table.

HISTOGRAM

Natural translates the HISTOGRAM statement into an SQL SELECT statement. The HISTOGRAM statement returns the number of rows in a table that have the same value in a specific column. The number of rows is returned in the Natural system variable *NUMBER.

Example:

Natural statements:

```
HISTOGRAM EMPLOYEES FOR AGE
OBTAIN AGE
```

Equivalent SQL statements:

```
SELECT AGE, COUNT(*) FROM EMPLOYEES
  GROUP BY AGE
  ORDER BY AGE
```

READ

Natural translates a READ statement into an SQL SELECT statement. Both READ PHYSICAL and READ LOGICAL statements can be used.

A row read with a READ LOGICAL statement (Example 1) cannot be updated or deleted. Natural translates a READ LOGICAL statement into the ORDER BY clause of an SQL SELECT statement, which produces a read-only result table.

A READ PHYSICAL statement (Example 2) can be updated or deleted. Natural translates it into a SELECT statement without an ORDER BY clause.

Example 1:

Natural statements:

```
READ PERSONNEL BY NAME
OBTAIN NAME FIRSTNAME DATEOFBIRTH
```

Equivalent SQL statement:

```
SELECT NAME, FIRSTNAME, DATEOFBIRTH FROM PERSONNEL
  WHERE NAME >= ' '
    ORDER BY NAME
```

Example 2:

Natural statements:

```
READ PERSONNEL PHYSICAL
OBTAIN NAME
```

Equivalent SQL statement:

```
SELECT NAME FROM PERSONNEL
```

When a `READ` statement contains a `WHERE` clause, Natural evaluates the `WHERE` clause after the rows have been selected according to the search criterion.

STORE

The `STORE` statement adds a row to a database table. It corresponds to the SQL `INSERT` statement.

Example:

Natural statement:

```
STORE RECORD IN EMPLOYEES
  WITH PERSONNEL_ID = '2112'
       NAME         = 'LIFESON'
       FIRST_NAME   = 'ALEX'
```

Equivalent SQL statement:

```
INSERT INTO EMPLOYEES (PERSONNEL_ID, NAME, FIRST_NAME)
  VALUES ('2112', 'LIFESON', 'ALEX')
```

UPDATE

The DML `UPDATE` statement updates a table row that has been read with a preceding `FIND`, `READ`, or `SELECT` statement. Natural translates the DML `UPDATE` statement into the SQL statement `UPDATE WHERE CURRENT OF` *`cursor-name`* (a positioned `UPDATE` statement), which means that only the last row that was read can be updated. In the case of nested loops, the last row in each nested loop can be updated.

UPDATE with FIND/READ

When a DML `UPDATE` statement is used after a Natural FIND statement, Natural translates the FIND statement into an SQL SELECT statement with a `FOR UPDATE OF` clause, and translates the DML `UPDATE` statement into an `UPDATE WHERE CURRENT OF` *`cursor-name`* statement.

Example:

```
FIND EMPLOYEES WITH SALARY < 5000
  ASSIGN SALARY = 6000
  UPDATE
```

Natural translates the Natural statements above into the following SQL statements and assigns a cursor name (for example, "CURSOR1"). The SELECT and UPDATE statements refer to the same cursor.

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY < 5000
  FOR UPDATE OF SALARY
UPDATE EMPLOYEES SET SALARY = 6000
  WHERE CURRENT OF CURSOR1
```

You cannot update a row read with a FIND SORTED BY or READ LOGICAL statement. For an explanation, see the FIND and READ statement descriptions above.

An END TRANSACTION or BACKOUT TRANSACTION statement releases data locked by an UPDATE statement.

UPDATE with SELECT

The DML UPDATE statement can be used after a SELECT statement only in the following case:

```
SELECT *
  INTO VIEW view-name
```

Natural rejects any other form of the SELECT statement used with the DML UPDATE statement. Natural translates the DML UPDATE statement into a non-cursor or „searched“ SQL UPDATE statement, which means than only an entire Natural view can be updated; individual columns cannot be updated.

In addition, the DML UPDATE statement can be used after a SELECT statement only in Natural structured mode, which has the following syntax:

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
```

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE NAME LIKE 'S%'
  OBTAIN NAME
```

```
    IF NAME = 'SMITH'
      ADD 1 TO AGE
    UPDATE
    END-IF
END-SELECT
```

In other respects, the DML `UPDATE` statement works with the `SELECT` statement the way it works with the Natural `FIND` statement (see ***UPDATE with FIND/READ*** above).

Natural SQL Statements

The SQL statements available within the Natural programming language comprise two different sets of statements: the common set and the extended set. On this platform, only the extended set is supported by Natural.

The common set can be handled by each SQL-eligible database system supported by Natural. It basically corresponds to the standard SQL syntax definitions. For a detailed description of the common set of Natural SQL statements, see *Common Set and Extended Set* (in the *Statements* documentation).

This section describes considerations and restrictions when using the common set of Natural SQL statements with Entire Access.

- DELETE
- INSERT
- PROCESS SQL
- SELECT
- UPDATE

DELETE

The Natural SQL `DELETE` statement deletes rows in a table without using a cursor.

Whereas Natural translates the DML `DELETE` statement into a positioned `DELETE` statement (that is, an SQL `DELETE WHERE CURRENT OF` *`cursor-name`* statement), the Natural SQL `DELETE` statement is a non-cursor or searched `DELETE` statement. A searched `DELETE` statement is a stand-alone statement unrelated to any `SELECT` statement.

INSERT

The INSERT statement adds rows to a table; it corresponds to the Natural STORE statement.

PROCESS SQL

The PROCESS SQL statement issues SQL statements in a *statement-string* to the database identified by a *ddm-name*.

Anmerkung: It is not possible to run database loops using the PROCESS SQL statement.

Parameters

Natural supports the INDICATOR and LINDICATOR clauses. As an alternative, the *statement-string* may include parameters. The syntax item *parameter* is syntactically defined as follows:

```
[ :U ] :host-variable
[ :G ]
```

A *host-variable* is a Natural program variable referenced in an SQL statement.

SET SQLOPTION option=value

With Entire Access, you can also specify SET SQLOPTION *option=value* as *statement-string*. This can be used to specify various options for accessing SQL databases. The options apply only to the database referenced by the PROCESS SQL statement.

Supported options are:

- DATEFORMAT
- DBPROCESS (for Sybase only)
- TIMEOUT (for Sybase only)
- TRANSACTION (for Sybase only)

DATEFORMAT

This option specifies the format used to retrieve SQL Date and Datetime information into Natural fields of type A. The option is obsolete if Natural fields of type D or T are used. A subset of the Natural date and time edit masks can be used:

YYYY	Year (4 digits)
YY	Year (2 digits)
MM	Month
DD	Day
HH	Hour
II	Minute
SS	Second

If the date format contains blanks, it must be enclosed in apostrophes.

Examples:

To use ISO date format, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION DATEFORMAT = YYYY-MM-DD >>
```

To obtain date and time components in ISO format, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION DATEFORMAT = 'YYYY-MM-DD HH:II:SS'    >>
```

The `DATEFORMAT` is evaluated only if data are retrieved from the database. If data are passed to the database, the conversion is done by the database system. Therefore, the format specified with `DATEFORMAT` should be a valid date format of the underlying database.

If no `DATEFORMAT` is specified for Natural fields,

- the default date format DD-MON-YY is used (where "MON" is a 3-letter abbreviation of the English month name) and

- the following default datetime formats are used:

Adabas D	YYYYMMDDHHIISS
DB2	YYYY-MM-DD-HH.II.SS
INFORMIX	YYYY-MM-DD HH:II:SS
ODBC	YYYY-MM-DD HH:II:SS
ORACLE	YYYYMMDDHHIISS
SYBASE DBLIB	YYYYMMDD HH:II:SS
SYBASE CTLIB	YYYYMMDD HH:II:SS
Microsoft SQL Server	YYYYMMDD HH:II:SS
other	DD-MON-YY

DBPROCESS

This option is valid for Sybase and Microsoft SQL Server databases only.

This option is used to influence the allocation of SQL statements to Sybase and Microsoft SQL Server `DBPROCESS`es. `DBPROCESS`es are used by Entire Access to emulate database cursors, which are not provided by the Sybase and Microsoft SQL Server DBlib interface.

Two values are possible:

MULTIPLE	With `DBPROCESS` set to `MULTIPLE`, each `SELECT` statement uses its own secondary `DBPROCESS`, whereas all other SQL statements are executed within the primary `DBPROCESS`. The value `MULTIPLE` therefore enables your application to execute further SQL statements, even if a database loop is open. It also allows nested database loops.
SINGLE	With `DBPROCESS` set to `SINGLE`, all SQL statements use the same (that is, the primary) `DBPROCESS`. It is therefore not possible to execute a new database statement while a database loop is active, because one `DBPROCESS` can only execute one SQL batch at a time. Since all statements are executed in the same (primary) `DBPROCESS`, however, this setting enables `SELECT`ions from non-shared temporary tables.

Anmerkungen:

1. The specified value can only be changed if no database loop is active.
2. As the `DBPROCESS` option only applies to the Sybase and Microsoft SQL Server DBlib interface, your application should use a central `CALLNAT` statement to change the value (at least for `SINGLE`), so that you can easily remove these calls once Sybase client libraries are supported. Your application should also use a central error handling that establishes the default setting (`MULTIPLE`).

TIMEOUT

This option is valid for Sybase and Microsoft SQL Server databases only.

With Sybase and Microsoft SQL Server, Entire Access uses a timeout technique to detect database-access deadlocks. The default timeout period is 8 seconds. With this option, you can change the duration of the timeout period (in seconds).

For example, to set the timeout period to 30 seconds, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION TIMEOUT = 30 >>
```

TRANSACTION

This option is valid for Sybase and Microsoft SQL Server databases only.

This option is used to enable or disable transaction mode. It becomes effective after the next `END TRANSACTION` or `BACKOUT TRANSACTION` statement.

If transaction mode is enabled (this is the default), Natural automatically issues all required statements to begin a transaction.

Examples:

To disable transaction mode, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION TRANSACTION = NO >>
...
END TRANSACTION
```

To enable transaction mode, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION TRANSACTION = YES >>
...
END TRANSACTION
```

SQLDISCONNECT

With Entire Access, you can also specify `SQLDISCONNECT` as the *statement-string*. In combination with the `SQLCONNECT` statement (see **below**), this statement can be used to access different databases by one application within the same session, by simply connecting and disconnecting as required.

A successfully performed `SQLDISCONNECT` statement clears the information previously provided by the `SQLCONNECT` statement; that is, it disconnects your application from the currently connected SQL database determined by the DBID of the DDM used in the `PROCESS SQL` statement. If no connection is established, the `SQLDISCONNECT` statement is ignored. It will fail if a transaction is open.

Anmerkung: If Natural reports an error in the `SQLDISCONNECT` statement, the connection status does not change. If the database reports an error, the connection status is undefined.

SQLCONNECT option=value

With Entire Access, you can also specify `SQLCONNECT` *option=value* as the *statement-string*. This statement can be used to establish a connection to an SQL database according to the DBID specified in the DDM addressed by the `PROCESS SQL` statement. The `SQLCONNECT` statement will fail if the specified connection is already established.

Supported options are:

- USERID
- PASSWORD
- OS_PASSWORD
- OS_USERID
- DBMS_PARAMETER

Anmerkungen:

1. If the `SQLCONNECT` statement fails, the connection status does not change.
2. If several options are specified, they must be separated by a comma.
3. The specified value can be either a character literal or a Natural variable of format A.
4. If Natural performs an implicit reconnect, because the connection to the database was lost, the values provided by the `SQLCONNECT` statement are used.

The options are evaluated as described below.

USERID and PASSWORD

Specifying `USERID` and `PASSWORD` for the database logon suppresses the default logon window and the evaluation of the environment variables `SQL_DATABASE_USER` and `SQL_DATABASE_PASSWORD`.

If only `USERID` is specified, `PASSWORD` is assumed to be blank, and vice versa.

If neither `USERID` nor `PASSWORD` is specified, default logon processing applies.

Anmerkung: With database systems that do not require user ID and password, a blank user ID and password can be specified to suppress the default logon processing.

OS_USERID and OS_PASSWORD

Specifying OS_PASSWORD and OS_USERID for the operating system logon suppresses the logon window and the evaluation of the environment variables SQL_OS_USER and SQL_OS_PASSWORD.

If only OS_USERID is specified, OS_PASSWORD is assumed to be blank, and vice versa.

If neither OS_USERID nor OS_PASSWORD is specified, default logon processing applies.

Anmerkung: With operating systems that do not require user ID and password, a blank user ID and password can be specified to suppress the default logon processing.

DBMS_PARAMETER

Specifying DBMS_PARAMETER dynamically overwrites the DBMS assignment in the Natural global configuration file.

Examples:

```
PROCESS SQL sql-ddm << SQLCONNECT USERID = 'DBA', PASSWORD = 'SECRET'    >>
```

This example connects to the database specified in the Natural global configuration file with user ID "DBA" and password "SECRET".

```
DEFINE DATA LOCAL
1 #UID (A20)
1 #PWD (A20)
END-DEFINE
INPUT 'Please enter ADABAS D user ID and password' / #UID / #PWD
PROCESS SQL sql-ddm << SQLCONNECT USERID = : #UID,
                       PASSWORD         = : #PWD,
                       DBMS_PARAMETER    = 'ADABASD:mydb'
                    >>
```

This example connects to the Adabas D database "mydb" with the user ID and password taken from the INPUT statement.

```
PROCESS SQL sql-ddm << SQLCONNECT USERID = ' ', PASSWORD = ' ',
                       DBMS_PARAMETER    = 'DB2:EXAMPLE' >>
```

This example connects to the DB2 database "EXAMPLE" without specifying user ID and password (since these are not required by DB2 which uses the operating system user ID).

SELECT

The `INTO` clause and scalar operators for the `SELECT` statement either are RDBMS-specific and do not conform to the standard SQL syntax definitions (the Natural common set), or impose restrictions when used with Entire Access.

Entire Access does not support the `INDICATOR` and `LINDICATOR` clauses in the `INTO` clause. Thus, Entire Access requires the following syntax for the `INTO` clause:

```
INTO [ parameter, ...          ]
     [ VIEW {view-name},...    ]
```

Anmerkung: The concatenation operator (`||`) does not belong to the common set and is therefore not supported by Entire Access.

SELECT SINGLE

The `SELECT SINGLE` statement provides the functionality of a non-cursor `SELECT` operation (singleton `SELECT`); that is, a `SELECT` statement that retrieves a maximum of one row without using a cursor.

This statement is similar to the Natural `FIND UNIQUE` statement. However, Natural automatically checks the number of rows returned. If more than one row is selected, Natural returns an error message.

If your RDBMS does not support dynamic execution of a non-cursor `SELECT` operation, the Natural `SELECT SINGLE` statement is executed like a set-level `SELECT` statement, which results in a cursor operation. However, Natural still checks the number of returned rows and issues an error message if more than one row is selected.

UPDATE

The Natural SQL `UPDATE` statement updates rows in a table without using a cursor.

Whereas Natural translates the DML `UPDATE` statement into a positioned `UPDATE` statement (that is, the SQL `DELETE WHERE CURRENT OF` *cursor-name* statement), the Natural SQL `UPDATE` statement is a non-cursor or searched `UPDATE` statement. A searched `UPDATE` statement is a stand-alone statement unrelated to any `SELECT` statement.

Flexible SQL

Flexible SQL allows you to use arbitrary RDBMS-specific SQL syntax extensions. Flexible SQL can be used as a replacement for any of the following syntactical SQL items:

- atom
- column reference
- scalar expression
- condition

The Natural compiler does not recognize the SQL text used in flexible SQL; it simply copies the SQL text (after substituting values for the host variables, which are Natural program variables referenced in an SQL statement) into the SQL string that it passes to the RDBMS. Syntax errors in flexible SQL text are detected at runtime when the RDBMS executes the string.

Note the following characteristics of flexible SQL:

- It is enclosed in "<<" and ">>" characters and can include arbitrary SQL text and host variables.
- Host variables must be prefixed by a colon (:).
- The SQL string can cover several statement lines; comments are permitted.

Flexible SQL can also be used between the clauses of a select expression:

```
SELECT selection
  << ... >>
  INTO ...
  FROM ...
  << ... >>
  WHERE ...
  << ... >>
  GROUP BY ...
  << ... >>
  HAVING ...
  << ... >>
  ORDER BY ...
  << ... >>
```

Examples:

```
SELECT NAME
FROM EMPLOYEES
WHERE << MONTH (BIRTH) >> = << MONTH (CURRENT_DATE) >>

SELECT NAME
FROM EMPLOYEES
WHERE << MONTH (BIRTH) = MONTH (CURRENT_DATE) >>

SELECT NAME
FROM EMPLOYEES
WHERE SALARY > 50000
<< INTERSECT
   SELECT NAME
   FROM EMPLOYEES
   WHERE DEPT = 'DEPT10'
>>
```

RDBMS-Specific Requirements and Restrictions

This section discusses restrictions and special requirements for Natural and some RDBMSs used with Entire Access.

The following topics are covered:

- Case-Sensitive Database Systems
- SYBASE and Microsoft SQL Server

Case-Sensitive Database Systems

In case-sensitive database systems, use lower-case characters for table and column names, as all names specified in a Natural program are automatically converted to lower-case.

Anmerkung: This restriction does not apply when you use flexible SQL.

SYBASE and Microsoft SQL Server

To execute SQL statements against SYBASE and Microsoft SQL Server, you must use one or more DBPROCESS structures. A DBPROCESS can execute SQL command batches.

A command batch is a sequence of SQL statements. Statements must be executed in the sequence in which they are defined in the command batch. If a statement (for example, a SELECT statement) returns a result, you must execute the statement first and then fetch the rows one by one. Once you execute the next statement from the command batch, you can no longer fetch rows from the previous query.

With SYBASE and Microsoft SQL Server, an application can use more than one DBPROCESS structure; therefore, it is possible to have nested queries if you use a separate DBPROCESS for each query. Because SYBASE and Microsoft SQL Server lock data for each DBPROCESS, however, an application that uses more than one DBPROCESS can deadlock itself. Natural times out in case of a deadlock.

The following topics are covered below:

- How Natural Statements are Converted to Database Calls
- Natural Restrictions with SYBASE and Microsoft SQL Server

How Natural Statements are Converted to Database Calls

Natural uses one DBPROCESS for each open query and another DBPROCESS for all other SQL statements (UPDATE, DELETE, INSERT, ...).

If a query is referenced by a positioned UPDATE or DELETE statement, Natural automatically appends the FOR BROWSE clause to the generated SELECT statement to allow UPDATEs while rows are being read.

For a positioned UPDATE or DELETE statement, the SYBASE dbqual function is used to generate the following search condition:

```
WHERE unique-index = value AND tsequal (timestamp,old-timestamp)
```

This search condition can be used to reselect the current row from the query. The tsequal function checks whether the row has been updated by another user.

Natural Restrictions with SYBASE and Microsoft SQL Server

The following restrictions apply when using Natural with SYBASE and Microsoft SQL Server.

Case-Sensitivity
SYBASE and Microsoft SQL Server are case-sensitive, and Natural passes parameters in lowercase. Thus, if your SYBASE and Microsoft SQL Server tables or fields are defined in uppercase or mixed case, you must use database `SYNONYM`s or Natural flexible SQL.

Positioned UPDATE and DELETE Statements
To support positioned `UPDATE` and `DELETE` statements, the table to be accessed must have a unique index and a timestamp column. In addition, the timestamp column must not be included in the select list of the query.

Querying Rows
SYBASE and Microsoft SQL Server lock pages, and locked pages are owned by `DBPROCESS` structures.

Pages locked by an active `DBPROCESS` cannot subsequently be read (by the same or another `DBPROCESS`) until the lock is released by an `END TRANSACTION` or `BACKOUT TRANSACTION` statement.

Therefore, if you have updated, inserted, or deleted a row in a table:

- Do not start a new `SELECT` (`FIND`, `READ`, ...) loop against the same table.
- Do not fetch additional rows from a query that references the same table if the `SELECT` statement has no `FOR BROWSE` clause.

Natural automatically appends the `FOR BROWSE` clause if the query is referenced by a positioned `UPDATE` or `DELETE` statement.

Transaction/Non-Transaction Mode
SYBASE and Microsoft SQL Server differentiate between transaction and non-transaction mode. In transaction mode, Natural connects to the database allowing `INSERT`s, `UPDATE`s and `DELETE`s to be issued; thus, commands that run in non-transaction mode, for example, `CREATE TABLE`, cannot be issued.

Stored Procedures
It is possible to use stored procedures in SYBASE and Microsoft SQL Server using the `PROCESS SQL` statement. However, the stored procedures must not contain

- commands that work only in non-transaction mode; or
- return values.

Data-Type Conversion

When a Natural program accesses data in a relational database, Entire Access converts RDBMS-specific data types to Natural data formats, and vice versa. The RDBMS data types and their corresponding Natural data formats are described in the *Editors* documentation under *Data Conversion for RDBMS* (in the section *DDM Editors*.

The date/time or datetime format specific to a particular database can be converted into the Natural formats D and T; see below.

Date/Time Conversion

The RDBMS-specific date/time or datetime format can be converted into the Natural formats D and T.

To use this conversion, you first have to edit the Natural DDM to change the date or time field formats from A(lphanumeric) to D(ate) or T(ime). The `SQLOPTION DATEFORMAT` is obsolete for fields with format D or T.

Anmerkung: Date or time fields converted to Natural D(ate)/T(ime) format may not be mixed with those converted to Natural A(lphanumeric) format.

- For update commands, Natural converts the Natural Date and Time format to the database-dependent representation of `DATE`/`TIME`/`DATETIME` to a precision level of seconds.
- For retrieval commands, Natural converts the returned database-dependent character representation to the internal Natural Date or Time format; see conversion tables below. The date component of Natural Time is not ignored and is initialized to 0000-01-02 (YYYY-MM-DD) if the RDBMS`s time format does not contain a date component.
- For Natural Date variables, the time portion is ignored and initialized to zero.
- For Natural Time variables, tenth of seconds are ignored and initialized to zero.

Conversion Tables

Adabas D

RDBMS Formats	Natural Date	Natural Time
DATE	YYYYMMDD	
TIME		00HHIISS

DB2

RDBMS Formats	Natural Date	Natural Time
DATE	YYYY-MM-DD	
TIME		HH.II.SS

INFORMIX

RDBMS Formats	Natural Date	Natural Time
DATETIME, year to day	YYYY-MM-DD	
DATETIME, year to second (other formats are not supported)		YYYY-MM-DD-HH:II:SS*

ODBC

RDBMS Formats	Natural Date	Natural Time
DATE	YYYY-MM-DD	
TIME		HH:II:SS

ORACLE

RDBMS Formats	Natural Date	Natural Time
DATE (ORACLE session parameter NLS_DATE_FORMAT is set to YYYYMMDDHH24MISS)	YYYYMMDD000000 (ORACLE time component is set to null for update commands and ignored for retrieval commands.)	YYYYMMDDHHIISS *

SYBASE

RDBMS Formats	Natural Date	Natural Time
DATETIME	YYYYMMDD	YYYYMMDD HH:II:SS *

* When comparing two time values, remember that the date components may have different values.

Microsoft SQL Server

RDBMS Formats	Natural Date	Natural Time
DATETIME	YYYYMMDD	YYYYMMDD HH:II:SS *

Obtaining Diagnostic Information about Database Errors

If the database returns an error while being accessed, you can call the non-Natural program `CMOSQERR` to obtain diagnostic information about the error, using the following syntax:

```
CALL 'CMOSQERR' parm1 parm2
```

The parameters are:

Parameter	Format/Length	Description
parm1	I4	The number of the error returned by the database.
parm2	A70	The text of the error returned by the database.

SQL Authorization

The Natural Configuration Utility allows you to add DBID specific settings of user IDs and passwords for automatic login to SQL databases. It distinguishes between operating system authentication and database authentication, depending on the current database system. If the **Auto login** flag in the **SQL Authorization** table is set for an SQL DBID then no interactive login prompt will pop up. The login values will be taken from this table row.

Please refer to *SQL Assignments* in the *Configuration Utility* documentation for a more detailed description of the SQL Authorization table.

31 Accessing Data in a Tamino Database

- Prerequisite 284
- DDM and View Definitions with Natural for Tamino 284
- Natural Statements for Tamino Database Access 288
- Natural for Tamino Restrictions 293

This document describes the different aspects of accessing a Tamino database with the Natural data manipulation language (DML).

For information about how to configure Natural to work with Tamino, see *Natural for Tamino* in the *Database Management System Interfaces* documentation.

Prerequisite

Tamino stores structured data-oriented XML documents in containers called doctypes. The doctypes are grouped logically together in so-called collections. Collections are stored in a Tamino database, which is the physical container of data.

The kind of data that can be stored in Tamino and that is to be accessed by Natural for Tamino must be defined in a Tamino XML Schema.

DDM and View Definitions with Natural for Tamino

This section describes the basic concepts of the Tamino XML schema language, Natural DDMs and view definitions and how they interact with Natural for Tamino.

The following topics are covered:

- Introducing Tamino XML Schema Language
- DDMs from Tamino
- Arrays in DDMs from Tamino
- Example of a DDM
- Definition of Views

Introducing Tamino XML Schema Language

The Tamino XML schema language is used to define a data type-oriented description of the structure of XML documents. In Tamino, a doctype represents a container for XML documents with the same root element and the same structure within a collection.

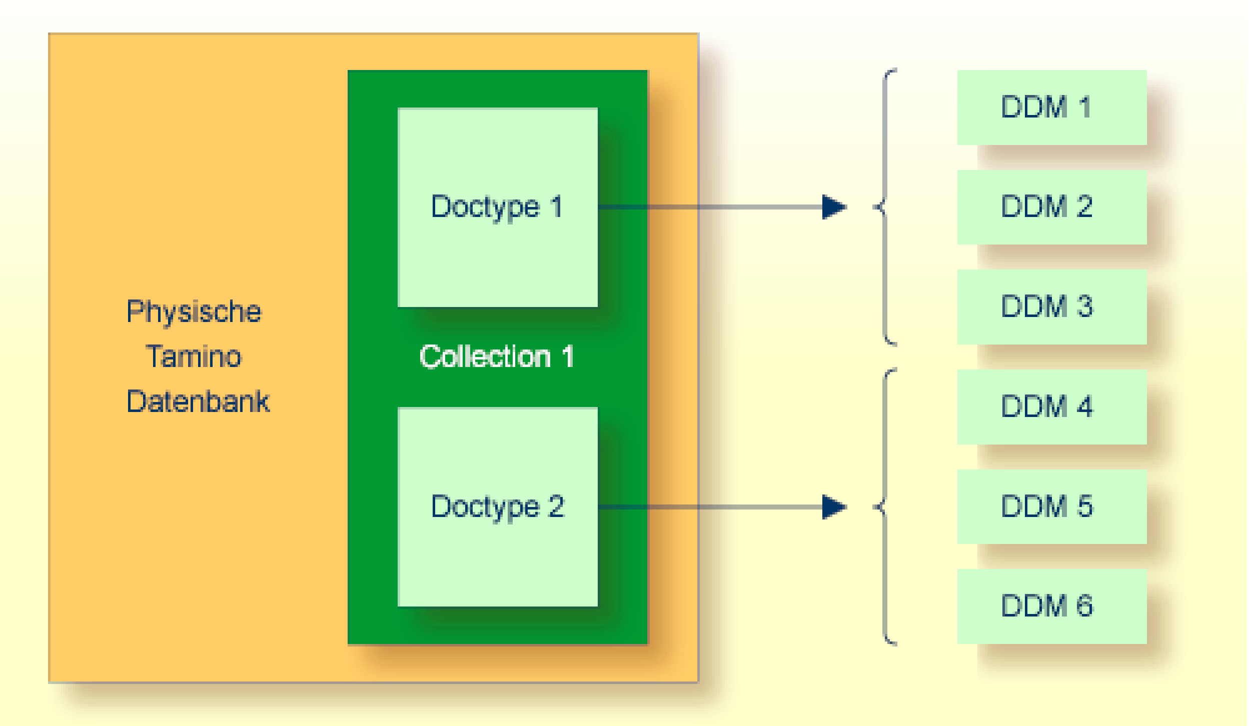
In Tamino, a collection is a container for a set of varying doctypes, so that a collection can be seen as the logical grouping of doctypes that belong together.

In a Tamino XML schema definition, a doctype is defined together with the collection in which it is contained. One Tamino XML schema can define more than one doctype and it can also define doctypes for more than one collection.

For more information on the Tamino XML schema language, refer to the Tamino documentation.

DDMs from Tamino

For Natural to be able to access a Tamino database, a logical connection between a Tamino doctype and the Natural data structures must be provided. Such a logical connection is called a DDM (data definition module).



A Natural DDM generated from a Tamino database is a representation of one doctype defined in one schema. The DDM contains information about the type of each data field and all the necessary structural information as defined in the corresponding Tamino XML schema. To generate a new DDM, the doctype must be selected from a list of all doctypes available in a given collection. Since one collection is bound to one Natural database ID (DBID), it is necessary to use a second DBID if a doctype from another collection is to be accessed.

A Tamino XML schema describes data and data structures in a very different way than with Natural data definitions. Therefore, specific mappings are introduced to derive a Natural data format from a Tamino XML schema data type.

You define DDMs with the Natural DDM editor. For more information about Tamino XML schema mapping, refer to *Data Conversion for Tamino* in the *DDM Editor* section of the *Editors* documentation.

For the field attributes defined in a DDM, refer to the *DDM Editor*, *Using the DDM Editor Window* section in the *Editors* documentation.

Arrays in DDMs from Tamino

If you define an XML element with a `maxOccurs` value greater than one in the Tamino XML Schema, then this element can occur as often as this value indicates. Such a construction is mapped either on a Natural static array definition or on a Natural X-Array definition. Depending on the type of the XML element you are dealing with, the following situations may occur:

- If the XML element is a `complexType` with `complexContent` (i.e. it is an element containing other elements) then the generated corresponding Natural group will be an indexed group.
- If the XML element is a `simpleType` (i.e. the element is holding data only) or a `complexType` with `simpleContent` (i.e. the element has only data and attributes but no other elements) then the generated Natural data field will be an array.

For further information about mapping `maxOccurs` definitions onto Natural arrays, see *Data Conversion for Tamino* in the *DDM Editor* section of the *Editors* documentation. The array boundaries or the kind of the array (static array or X-Array) can be adapted in a corresponding view definition as usual.

Example of a DDM

This is an example of an EMPLOYEES DDM generated from a Tamino XML Schema definition.

The schema can, for example, be defined with the Natural demo application SYSEXINS:

```
DB: 00250 FILE: 00001  - EMPLOYEES-XML
TYPE: XML
COLLECTION: NATDemoData
SCHEMA: Employee
DOCTYPE: Employee
NAMESPACE-PREFIX: xs
NAMESPACE-URI: http://www.w3.org/2001/XMLSchema
T L  Name                             F Leng       D Remark
- -- -------------------------------- - ---------- - -----------
G  1 EMPLOYEE
     FLAGS=MULT_REQUIRED,MULT_ONCE
     TAG=Employee
     XPATH=/Employee
G  2 GROUP$1
     FLAGS=GROUP_ATTRIBUTES
   3 PERSONNEL-ID                     A          8 D xs:string
     FLAGS=ATTR_REQUIRED
     TAG=@Personnel-ID
     XPATH=/Employee/@Personnel-ID
G  2 GROUP$2
     FLAGS=GROUP_SEQUENCE,MULT_REQUIRED,MULT_ONCE
G  3 FULL-NAME
     FLAGS=MULT_OPTIONAL
     TAG=Full-Name
```

```
     XPATH=/Employee/Full-Name
G  4 GROUP$3
     FLAGS=GROUP_SEQUENCE,MULT_REQUIRED,MULT_ONCE
   5 FIRST-NAME                    A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=First-Name
     XPATH=/Employee/Full-Name/First-Name
   5 MIDDLE-NAME                   A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Middle-Name
     XPATH=/Employee/Full-Name/Middle-Name
   5 MIDDLE-I                      A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Middle-I
     XPATH=/Employee/Full-Name/Middle-I
   5 NAME                          A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Name
     XPATH=/Employee/Full-Name/Name
       . . .
   3 LANG                          A          3   xs:string
     FLAGS=ARRAY,MULT_OPTIONAL
     OCC=1:4
     TAG=Lang
     XPATH=/Employee/Lang
```

Definition of Views

In order to work with Tamino database fields in a Natural program, you must specify the required fields of the DDM in a Natural *view-definition* (see the `DEFINE DATA` statement). Normally, a view is a special subset of the complete data structure as defined in the DDM.

Tamino XML Schema->Natural for Tamino DDM->Natural *view-definition*

If the view is used to store XML objects, it has to contain all fields that are required to a generate documents that are valid according to the corresponding Tamino XML schema definition.

A view for the EMPLOYEES-XML DDM, where one of the view fields is a static array, might look like this:

```
DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-XML
02 NAME
02 CITY
02 LANG (1:4)
END-DEFINE
```

Natural Statements for Tamino Database Access

The Natural DML statements which are provided for Tamino access can be subdivided into two categories:

- pure retrieval statements;
- database modification statements.

The Natural system variable `*ISN` is mapped on the Tamino `ino:id`.

Natural for Tamino Retrieval Statements

The following Natural statements can be used for database retrieval:

- `FIND`

  This statement is used to select those records from a database which meet a specified search criterion.
- `GET`

  This statement is used to select one special record with its unique id from the database.
- `READ`

  This statement is used to select a range of records from a database in a specified sequence.

Not all of the possible options and all of the possible clauses of the retrieval statements can be used for Tamino access. Please read the appropriate section in the *Statements* documentation for a detailed description.

All statements are internally realized with the Tamino `_xquery` command verb. Statement clauses are mapped to corresponding Tamino XQuery expressions, e.g. search criteria are mapped to Tamino XQuery comparison expressions, sequence specifications are mapped to Tamino XQuery ordering expressions with sort direction.

The result set for the `FIND` and `READ` statements is determined at start of the loop and remains unchanged throughout the loop.

The following is an example of reading a set of employee records from a Tamino database where one view field is an array:

```
* READ 5 RECORDS DESCENDING CONTAINING A
* STATIC ARRAY IN THE VIEW DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-TAMINO
02 NAME
02 CITY
02 LANG (1:4)
END-DEFINE
*
READ(5) VW DESCENDING BY NAME = 'MAYER'
  DISPLAY NAME CITY LANG(*)
END-READ
*
END
```

Natural for Tamino Database Modification Statements

The following database modification statements are provided for use with Natural for Tamino:

- `STORE`

  This statement is used for inserting a new XML document into the database.

- `DELETE`

  This statement is used for deleting a document from the database. The `DELETE` statement implements a positioned delete.

For a detailed description of the statements, see the appropriate sections of the *Statements* documentation.

The `DELETE` statement is internally realized with the Tamino `_delete` command verb using the current `ino:id`, and the `STORE` statement is implemented with the `_process` command verb.

Example:

The following example program stores a new employee record with some data in the database:

```
* STORE NEW EMPLOYEE
DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-TAMINO
02 PERSONNEL-ID
02 NAME
02 CITY
02 LANG (1:3)
END-DEFINE
*
* FILL VIEW
PERSONNEL-ID := '1230815'
NAME         := 'KENT'
CITY         := 'ROME'
LANG(1)      := 'ENG'
LANG(2)      := 'GER'
LANG(3)      := 'SPA'
*
* STORE VIEW
STORE RECORD IN VW
*
COMMIT
*
END
```

If the Tamino XML Schema defines data structures for a doctype as being mandatory, then these data structures must also be filled in the view before a `STORE` statement is issued, otherwise this will result in a Tamino error.

Natural for Tamino Logical Transaction Handling

Natural performs database modification operations based on transactions, which means that all database modification requests are processed in logical transaction units. A logical transaction is the smallest unit of work (as defined by you) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

A logical transaction may consist of one or more modification statements (`DELETE`, `STORE`) involving one or more doctypes in the database. A logical transaction may also span multiple Natural programs.

A logical transaction begins when a database modification statement is issued. Natural does this automatically. For example, if a `FIND` loop contains a `DELETE` statement. The end of a logical transaction is determined by an `END TRANSACTION` statement in the program. This statement ensures that all modifications within the transaction have been successfully applied.

Natural for Tamino Error Handling

In addition to Natural's standard error messages there are two special error codes which provide additional information via a sub-error code.

NAT8400 **Tamino error ... occurred**

Erläuterung For this special error an additional sub-code number is shown. This number refers to a Tamino error message. Please see the Tamino *Messages and Codes* documentation. The user exit USR6007 in library SYSEXT is provided for obtaining diagnostic information in case a NAT8400 error occurs. Here is an example of usage:

```
DEFINE DATA LOCAL
 01 VW VIEW OF EMPLOYEES-TAMINO
  02 NAME
  02 CITY
 01 TAMINO_PARMS
  02 TAMINO_ERROR_NUM        (I4)   /* Error number of Tamino error
  02 TAMINO_ERROR_TEXT       (A70)  /* Tamino error text
  02 TAMINO_ERROR_LINE       (A253) /* Tamino error message line
END-DEFINE
*
NAME := 'MEYER'
CITY := 'BOSTON'
STORE VW
*
ON ERROR
 IF *ERROR EQ 8400     /* in case of error 8400 obtain diagnostic
information
  CALLNAT 'USR6007N' TAMINO_PARMS
  PRINT 'Error 8400 occurred:'
  PRINT 'Error Number:' TAMINO_ERROR_NUM
  PRINT 'Error Text  :' TAMINO_ERROR_TEXT
  PRINT 'Error Line  :' TAMINO_ERROR_LINE
 END-IF
END-ERROR
*
END
```

NAT8411 **HTTP request failed with response code...**

Erläuterung The error code from the HTTP server is delivered as additional information. Siehe auch `REQUEST DOCUMENT` statement, *Overview of Response Numbers for HTTP/HTTPs Requests*.

Example of Natural for Tamino Interacting with a SQL Database

This is a more sophisticated example of Natural for Tamino interacting with an SQL database; it retrieves data from a Tamino database and inserts or updates the corresponding row in an appropriate table in a SQL database.

```
*
* TAMINO DB --> SQL RDBMS EXAMPLE
*
DEFINE DATA LOCAL
* DEFINE VIEW FOR TAMINO
01 VW-TAMINO VIEW OF EMPLOYEES-TAMINO
02 PERSONNEL-ID
02 NAME
02 CITY
* DEFINE VIEW FOR SQL DATABASE
01 VW-SQL VIEW OF EMPLOYEES-SQL
02 PERSONNEL_ID
02 NAME
02 CITY
END-DEFINE
*
* OPEN A TAMINO LOGICAL READ LOOP
*
TAMINO. READ VW-TAMINO BY NAME
*
* SEARCH RECORD IN SQL DATABASE AND
* INSERT A NEW RECORD IF NOT FOUND OR
* UPDATE THE EXISTING ONE WITH THE DATA
* FROM TAMINO DB
SQL. FIND(1) VW-SQL WITH PERSONNEL_ID = PERSONNEL-ID (TAMINO.)
       IF NO RECORDS FOUND
         PERSONNEL_ID := PERSONNEL-ID (TAMINO.)
         NAME         := NAME         (TAMINO.)
         CITY         := CITY         (TAMINO.)
         STORE VW-SQL
         ESCAPE BOTTOM
       END-NOREC
       PERSONNEL_ID := PERSONNEL-ID (TAMINO.)
       NAME         := NAME         (TAMINO.)
       CITY         := CITY         (TAMINO.)
       UPDATE
     END-FIND
*
END-READ
*
END TRANSACTION
*
END
```

Natural for Tamino Restrictions

There are restrictions concerning the scope of the Tamino XML Schema language that can be used for creating schemas for Natural for Tamino DDM generation:

- Only Tamino XML Schema language constructors and attributes (as mentioned in *Tamino XML Schema Constructors* in the *DDM Editor* section of the *Editors* documentation) are supported by Natural for Tamino. Other constructors such as `xs:any`, `xs:anyAttribute` cannot be applied in Tamino XML Schemas if you wish to use them together with Natural for Tamino.
- The functionality of `xs:import` is not supported by Natural for Tamino. This means that external schema components must not be referenced in a Tamino XML Schema suitable for usage together with Natural. In other words, a doctype definition in a Tamino XML Schema must resolve all references within this Tamino XML Schema itself if you are planning to use it together with Natural for Tamino.
- The attribute `mixed` of the constructor `xs:complexType` is only supported with its default value "false". Natural for Tamino does not support mixed-content document definitions (as set with the specification `mixed="true"`). Using `mixed="true"` will result in an error during DDM generation.
- The level of nested structures in a Natural for Tamino DDM is limited to 99. A new DDM level is generated whenever one of the following constructors occurs in the Tamino XML Schema:

  ```
  xs:element
  xs:attribute
  xs:choice
  xs:all
  xs:sequence
  ```

- Recursively defined structures in a Tamino XML Schema cannot be used together with Natural for Tamino.
- The Tamino XML Schema language constructor `xs:choice` is mapped on a Natural group containing all alternatives of the choice. To restrict processing to one particular choice, an appropriate view with the required choice has to be created.
- Natural for Tamino only supports „closed content validation mode“. Tamino XML Schemas with „open content validation mode“ cannot be used together with Natural for Tamino.
- For the Tamino XML Schema language constructors `xs:choice`, `xs:sequence` and `xs:all`, a value greater than 1 of the attribute `maxOccurs` cannot be handled in the Natural data structures. Hence a value greater than 1 will always lead to an error during DDM generation.
- Natural for Tamino can handle only Tamino objects that are defined with a Tamino XML Schema as a subset of the W3C schema. Especially Natural for Tamino does not support non-XML (`tsd:nonXML`) data or instances without a defined schema (`ino:etc`).

32 Steuerung der Ausgabe von Daten

Dieser Teil beschreibt, wie Sie vorgehen müssen, wenn ein Natural-Programm mehrere Reports erzeugen soll. Außerdem behandelt es verschiedene Möglichkeiten, wie Sie die Form eines mit Natural erzeugten Ausgabe-Reports, d.h. die Art und Weise, in der die Daten angezeigt werden, beeinflussen können.

Folgende Themen werden behandelt:

- **Report-Spezifikation — (*rep*)-Notation**
- **Layout einer Ausgabeseite**
- **Statements DISPLAY und WRITE**
- **Index-Notation für multiple Felder und Periodengruppen**
- **Seitenüberschriften, Seitenvorschübe und Leerzeilen**
- **Spaltenüberschriften**
- **Parameter zur Beeinflussung der Ausgabe von Feldern**
- **Editiermasken - der EM-Parameter**
- **Unicode-Editiermasken - der EMU-Parameter**
- **Vertikale Ausgaben**

33 Report-Spezifikation — (rep)-Notation

- Report-Spezifikationen benutzen 298
- Betroffene Statements 298
- Beispiele für Report-Spezifikation 298

`(rep)` ist der Ausgabereport-Identifikator, für den ein Statement anwendbar ist.

Report-Spezifikationen benutzen

Wenn ein Natural-Programm mehrere Reports erzeugen soll, muss die Notation (*rep*) bei jedem Ausgabe-Statement angegeben werden (siehe *Betroffene Statements*, weiter unten), das zum Erzeugen von Ausgaben für einen Report außer dem ersten Report (Report 0) benutzt werden soll.

Es kann ein Wert von 0 – 31 angegeben werden.

Der Wert für `(rep)` kann auch ein logischer Name sein, der mittels des `DEFINE PRINTER`-Statements zugewiesen wurde, siehe ***Beispiel 2*** weiter unten.

Betroffene Statements

Die Notation `(rep)` kann mit den folgenden Ausgabe-Statements benutzt werden:

`AT END OF PAGE` | `AT TOP OF PAGE` | `DISPLAY` | `EJECT` | `FORMAT` | `NEWPAGE` | `PRINT` | `SKIP` | `SUSPEND IDENTICAL SUPPRESS` | `WRITE` | `WRITE TITLE` | `WRITE TRAILER`

Beispiele für Report-Spezifikation

Beispiel 1 – Mehrere Reports

```
DISPLAY (1) NAME ...
WRITE (4) NAME ...
```

Beispiel 2 – Logische Namen benutzen

```
DEFINE PRINTER (LIST=5) OUTPUT 'LPT1'
WRITE (LIST) NAME ...
```

34 Layout einer Ausgabeseite

- Statements mit Auswirkungen auf das Aussehen eines Report-Layouts 300
- Allgemeines Layout-Beispiel 301

Dieses Kapitel gibt eine Übersicht über die Statements, die zur Definition eines spezifischen Layouts für einen Report benutzt werden können.

Statements mit Auswirkungen auf das Aussehen eines Report-Layouts

Folgende Statements haben Auswirkungen auf das Aussehen einer Ausgabe:

Statement	Funktion
`WRITE TITLE`	Mit diesem Statement können Sie eine Seiten-Kopfzeile angeben, d.h. Text, der am Anfang einer Seite ausgegeben werden soll. Standardmäßig sind Seiten-Kopfzeilen zentriert und nicht unterstrichen.
`WRITE TRAILER`	Mit diesem Statement können Sie eine Seiten-Fußzeile angeben, d.h. Text, der am Ende einer Seite ausgegeben werden soll. Standardmäßig sind Seiten-Fußzeilen zentriert und nicht unterstrichen.
`AT TOP OF PAGE`	Mit diesem Statement können Sie eine Verarbeitung angeben, die immer dann ausgeführt werden soll, wenn eine neue Ausgabeseite erzeugt wird. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese unter der Seiten-Kopfzeile ausgegeben.
`AT END OF PAGE`	Mit diesem Statement können Sie eine Verarbeitung angeben, die immer dann ausgeführt werden soll, wenn eine Seitenende-Bedingung vorliegt. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese unter der (mit dem `WRITE TRAILER`-Statement erzeugten) Seiten-Fußzeile ausgegeben.
`AT START OF DATA`	Mit diesem Statement können Sie eine Verarbeitung angeben, die ausgeführt werden soll, nachdem in einer Datenbank-Verarbeitungsschleife der erste Datensatz gelesen worden ist. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese vor dem ersten Feldwert ausgegeben.
AT END OF DATA	Mit diesem Statement können Sie eine Verarbeitung angeben, die ausgeführt werden soll, nachdem in einer Datenbank-Verarbeitungsschleife alle Datensätze verarbeitet worden sind. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese unmittelbar nach dem letzten Feldwert ausgegeben.
`DISPLAY` / `WRITE`	Mit diesen Statements steuern Sie die Art, in der gelesene Feldwerte ausgegeben werden. Siehe Abschnitt ***Statements DISPLAY und WRITE***.

Die Statements `AT START OF DATA` und `AT END OF DATA` sind im Kapitel *Datenbankzugriffe*, ***AT START/END OF DATA Statements***, beschrieben. Die anderen oben aufgeführten Statements sind in den folgenden Abschnitten des vorliegenden Dokuments beschrieben.

Allgemeines Layout-Beispiel

Das folgende Beispiel-Programm veranschaulicht die allgemeine Form einer Ausgabeseite:

```
** Example 'OUTPUX01': Several sections of output
************************************************************************
DEFINE DATA LOCAL
1 EMP-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
END-DEFINE
*
WRITE TITLE   '********** Page Title **********'
WRITE TRAILER '********** Page Trailer **********'
*
AT TOP OF PAGE
  WRITE '===== Top of Page ====='
END-TOPPAGE
AT END OF PAGE
  WRITE '===== End of Page ====='
END-ENDPAGE
*
READ (10) EMP-VIEW BY NAME
  /*
  DISPLAY NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD)
  /*
  AT START OF DATA
    WRITE '>>>>> Start of Data >>>>>'
  END-START
  AT END OF DATA
    WRITE '<<<<< End of Data <<<<<'
  END-ENDDATA
END-READ
END
```

Ausgabe des Programms `OUTPUX01`:

```
                    ********** Page Title **********
===== Top of Page =====
        NAME              FIRST-NAME         DATE
                                              OF
                                             BIRTH
-------------------- -------------------- ----------

>>>>> Start of Data >>>>>
```

```
ABELLAN              KEPA                 1961-04-08
ACHIESON             ROBERT               1963-12-24
ADAM                 SIMONE               1952-01-30
ADKINSON             JEFF                 1951-06-15
ADKINSON             PHYLLIS              1956-09-17
ADKINSON             HAZEL                1954-03-19
ADKINSON             DAVID                1946-10-12
ADKINSON             CHARLIE              1950-03-02
ADKINSON             MARTHA               1970-01-01
ADKINSON             TIMMIE               1970-03-03
<<<<< End of Data <<<<<
                  ********** Page Trailer **********
===== End of Page =====
```

35 Statements DISPLAY und WRITE

- Das DISPLAY-Statement 304
- Das WRITE-Statement 305
- Beispiel für ein DISPLAY-Statement 306
- Beispiel für ein WRITE-Statement 307
- Spaltenabstand - der SF-Parameter und die Notation nX 307
- Tabulator-Notation nT 309
- Zeilenvorschub — die Schrägstrich-Notation (/) 309
- Weitere Beispiele für DISPLAY- und WRITE-Statements 312

Dieses Kapitel beschreibt, wie Sie die Statements DISPLAY und WRITE zur Ausgabe von Daten und zur Steuerung der Art und Weise benutzen, in der die Informationen ausgegeben werden.

Das DISPLAY-Statement

Das DISPLAY-Statement erzeugt eine Ausgabe in Spaltenform; d.h. die Werte eines Feldes werden jeweils in einer Spalte untereinander ausgegeben. Wenn mehrere Felder ausgegeben werden, d.h. wenn mehrere Spalten erzeugt werden, werden diese Spalten nebeneinander ausgegeben.

Die Reihenfolge, in der die Felder ausgegeben werden, bestimmen Sie durch die Reihenfolge, in der Sie die Feldnamen im DISPLAY-Statement angeben.

Das DISPLAY-Statement im folgenden Programm zeigt für jede Person zuerst die Personalnummer (PERSONNEL-ID) an, dann den Nachnamen (NAME) und zuletzt die Tätigkeitsbezeichnung (JOB-TITLE):

```
** Example 'DISPLX01': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END
```

Ausgabe des Programms DISPLX01:

```
Page      1                                                04-11-11  14:15:54

PERSONNEL         NAME                 CURRENT
   ID                                 POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Um die Reihenfolge der Spalten in der Ausgabe zu ändern, ändern Sie einfach die Reihenfolge der Feldnamen im DISPLAY-Statement. Falls Sie beispielsweise zuerst die Nachnamen, dann die Tätigkeitsbezeichnungen und zuletzt die Personalnummern ausgeben möchten, müsste das entsprechende DISPLAY-Statement folgendermaßen aussehen:

```
** Example 'DISPLX02': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY NAME JOB-TITLE PERSONNEL-ID
END-READ
END
```

Ausgabe des Programms DISPLX02:

```
Page      1                                            04-11-11  14:15:54

        NAME                    CURRENT          PERSONNEL
                               POSITION             ID
-------------------- ------------------------- ---------

GARRET               TYPIST                    30020013
TAILOR               WAREHOUSEMAN              30016112
PIETSCH              SECRETARY                 20017600
```

Über jeder Spalte wird eine Spaltenüberschrift ausgegeben. Verschiedene Möglichkeiten, die Ausgabe dieser Überschriften zu beeinflussen, sind weiter unten im Abschnitt *Spaltenüberschriften* beschrieben.

Das WRITE-Statement

Das WRITE-Statement wird zur Erzeugung unformatierter (d.h. nicht in Spalten unterteilter) Ausgaben benutzt. Im Gegensatz zum DISPLAY-Statement gilt für das WRITE-Statement Folgendes:

- Es führt, wenn nötig, einen automatischen Zeilenvorschub aus; d.h. ein Feld oder Textelement, das nicht mehr in eine Zeile passt, wird automatisch in der nächsten Zeile ausgegeben.
- Es erzeugt keine Spaltenüberschriften.

- Bei Werten eines multiplen Feldes werden diese nicht untereinander sondern nebeneinander ausgegeben.

Die beiden Beispielprogramme auf der folgenden Seite veranschaulichen die grundsätzlichen Unterschiede zwischen `DISPLAY`- und `WRITE`-Statement.

Sie können beide Statements auch miteinander kombinieren; diese Möglichkeit ist unter *Vertikale Ausgabe von Feldwerten* im Abschnitt ***Vertikale Ausgabe durch Kombination von DISPLAY und WRITE*** beschrieben.

Beispiel für ein DISPLAY-Statement

```
** Example 'DISPLX03': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:3)
END-DEFINE
*
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME SALARY (1:3)
END-READ
END
```

Ausgabe des Programms `DISPLX03`:

```
Page      1                                                 04-11-11  14:15:54

        NAME                 FIRST-NAME        ANNUAL
                                               SALARY
-------------------- -------------------- ----------

JONES                VIRGINIA                   46000
                                                42300
                                                39300
JONES                MARSHA                     50000
                                                46000
                                                42700
```

Beispiel für ein WRITE-Statement

```
** Example 'WRITEX01': WRITE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:3)
END-DEFINE
*
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME FIRST-NAME SALARY (1:3)
END-READ
END
```

Ausgabe des Programms WRITEX01:

```
Page      1                                          04-11-11  14:15:55

JONES               VIRGINIA                   46000      42300      39300
JONES               MARSHA                     50000      46000      42700
```

Spaltenabstand - der SF-Parameter und die Notation nX

Standardmäßig sind die mit einem DISPLAY-Statement ausgegebenen Spalten jeweils durch *eine* Leerstelle voneinander getrennt.

Mit dem Session-Parameter SF (Spacing Factor) können Sie angeben, wieviele Leerstellen zwischen den Spalten einer DISPLAY-Ausgabe eingefügt werden sollen. Sie können die Anzahl der Leerstellen auf einen Wert von 1 bis 30 setzen.

Der Parameter kann in einem FORMAT-Statement angegeben werden und gilt dann für den ganzen Report. Oder er kann in einem DISPLAY-Statement angegeben werden, und zwar auf Statement-Ebene, aber nicht auf Element-Ebene.

Mit der Notation *n*X können Sie die Anzahl der Leerstellen (*n*) zwischen zwei bestimmten Spalten angeben. Eine *n*X-Notation hat Vorrang vor einer SF-Parameterangabe.

Beispiel:

```
** Example 'DISPLX04': DISPLAY (with nX)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
FORMAT SF=3
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME 5X JOB-TITLE
END-READ
END
```

Ausgabe des Programms `DISPLX04`:

Das obige Beispielprogramm erzeugt folgende Ausgabe, wobei die ersten beiden Spalten aufgrund des `SF`-Parameters im `FORMAT`-Statement durch 3 Leerstellen voneinander getrennt sind, während die zweite und dritte Spalte aufgrund der Notation `5X` im `DISPLAY`-Statement durch 5 Leerstellen voneinander getrennt sind:

```
Page      1                                                  04-11-11  14:15:54

PERSONNEL           NAME                         CURRENT
   ID                                            POSITION
---------   --------------------     -------------------------

30020013    GARRET                   TYPIST
30016112    TAILOR                   WAREHOUSEMAN
20017600    PIETSCH                  SECRETARY
```

Die Notation *nX* kann auch in einem `WRITE`-Statement verwendet werden, um Leerstellen zwischen Ausgabe-Elementen einzufügen:

```
WRITE PERSONNEL-ID 5X NAME 3X JOB-TITLE
```

Mit dem obigen Statement werden zwischen den Feldern `PERSONNEL-ID` und `NAME` 5 Leerstellen und zwischen `NAME` und `JOB-TITLE` 3 Leerstellen eingefügt.

Tabulator-Notation nT

Mit der Tabulator-Notation *n*T, die im DISPLAY- und im WRITE-Statement verwendet werden kann, können Sie die Spalte *n* bestimmen, ab der ein Ausgabe-Element ausgegeben werden soll.

```
** Example 'DISPLX05': DISPLAY (with nT)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 5T NAME 30T FIRST-NAME
END-READ
END
```

Ausgabe des Programms DISPLX05:

Das obige Programm erzeugt folgende Ausgabe, wobei das Feld NAME ab Spalte 5 (vom linken Seitenrand aus gezählt) und das Feld FIRST-NAME ab Spalte 30 ausgegeben wird:

```
Page      1                                                    04-11-11  14:15:54

            NAME                      FIRST-NAME
    --------------------     --------------------

    JONES                    VIRGINIA
    JONES                    MARSHA
    JONES                    ROBERT
```

Zeilenvorschub — die Schrägstrich-Notation (/)

Mit einem Schrägstrich (/) in einem DISPLAY- oder WRITE-Statement bewirken Sie einen Zeilenvorschub.

- Bei einem DISPLAY-Statement bewirkt ein Schrägstrich einen Zeilenvorschub *zwischen Feldern* und *innerhalb von Text*.
- Bei einem WRITE-Statement bewirkt ein Schrägstrich nur *zwischen Feldern* einen Zeilenvorschub; innerhalb von Text wird er wie ein gewöhnliches Textzeichen behandelt.

Zwischen Feldern muss dem Schrägstrich je ein Leerzeichen vor- und nachgestellt werden.

Für mehrfachen Zeilenvorschub geben Sie mehrere Schrägstriche an.

Beispiel 1 - Zeilenvorschub bei einem DISPLAY-Statement:

```
** Example 'DISPLX06': DISPLAY (with slash '/')
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT
END-READ
END
```

Ausgabe des Programms `DISPLX06`:

Das obige `DISPLAY`-Statement erzeugt einen Zeilenvorschub nach jedem Wert des Feldes `NAME` sowie innerhalb des Textes `DEPART-MENT`:

```
Page      1                                                  04-11-11  14:15:54

        NAME           DEPART-
     FIRST-NAME         MENT
-------------------- -------

JONES                SALE
VIRGINIA
JONES                MGMT
MARSHA
JONES                TECH
ROBERT
```

Beispiel 2 - Zeilenvorschub bei einem WRITE-Statement:

```
** Example 'WRITEX02': WRITE (with line advance)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
```

```
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT //
END-READ
END
```

Ausgabe des Programms `WRITEX02`:

Das obige `WRITE`-Statement erzeugt einen Zeilenvorschub nach jedem Wert des Feldes NAME und einen doppelten Zeilenvorschub nach jedem Wert des Feldes `DEPARTMENT`, aber keinen innerhalb des Textes `DEPART-/MENT`:

```
Page      1                                                  04-11-11  14:15:55

JONES
VIRGINIA             DEPART-/MENT SALE


JONES
MARSHA               DEPART-/MENT MGMT


JONES
ROBERT               DEPART-/MENT TECH
```

Beispiel 3 - Zeilenvorschub in DISPLAY- und WRITE-Statements:

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
```

```
          'HOME/CITY' CITY
          'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX21`:

```
14:15:54.6     PEOPLE LIVING IN SALT LAKE CITY                     PAGE:      1
               AS OF 11/11/2004


-------------------------------------------------------------------------------
        NAME                 HOME                 STREET
     FIRST-NAME              CITY               OR BOX NO.
-------------------- -------------------- --------------------

ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY

SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                   REGISTER OF
                                 SALT LAKE CITY
-------------------------------------------------------------------------------
```

Weitere Beispiele für DISPLAY- und WRITE-Statements

Siehe die folgenden Beispiel-Programme:

- *DISPLX13 - DISPLAY-Statement (zum Vergleich mit WRITEX08 mit WRITE)*
- *WRITEX08 - WRITE-Statement (zum Vergleich mit DISPLX13 mit DISPLAY)*
- *DISPLX14 - DISPLAY-Statement (mit AL, SF und nX)*
- *WRITEX09 - WRITE-Statement (in Kombination mit AT END OF DATA)*

36 Index-Notation für multiple Felder und Periodengruppen

- Index-Notation benutzen 314
- Beispiel für Index-Notation im DISPLAY-Statement 314
- Beispiel für Index-Notation im WRITE-Statement 315

Dieses Kapitel beschreibt, wie Sie die Index-Notation `(n:n)` benutzen können, um anzugeben, wieviele Werte eines multiplen Feldes oder wieviele Ausprägungen einer Periodengruppe ausgegeben werden sollen.

Index-Notation benutzen

Mit einer Index-Notation `(n:n)` können Sie angeben, wieviele Werte eines multiplen Feldes bzw. wieviele Ausprägungen einer Periodengruppe ausgegeben werden sollen.

Beispiel: Das Feld `INCOME` im DDM `EMPLOYEES` ist eine Periodengruppe und enthält das jährliche Einkommen eines Mitarbeiters für jedes Jahr der Betriebszugehörigkeit.

Die Daten werden in chronologischer Reihenfolge gespeichert, wobei das Einkommen des jeweils letzten Jahres in der Ausprägung `1` zu finden ist.

Wollen Sie das Jahreseinkommen eines Mitarbeiters in den letzten drei Jahren angezeigt bekommen, d.h. Ausprägungen `1` bis `3`, fügen Sie im `WRITE`- bzw. `DISPLAY`-Statement hinter dem betreffenden Feldnamen die Notation `(1:3)` ein (wie im folgenden Beispielprogramm gezeigt).

Beispiel für Index-Notation im DISPLAY-Statement

```
** Example 'DISPLX07': DISPLAY (with index notation)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 INCOME  (1:3)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME INCOME (1:3)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX07`:

Wenn mehrere Werte eines multiplen Feldes über ein `DISPLAY`-Statement ausgegeben werden, werden diese, wie Sie sehen, untereinander ausgegeben:

```
Page      1                                                   04-11-11  14:15:54

PERSONNEL         NAME                     INCOME
   ID
                               CURRENCY   ANNUAL     BONUS
                                 CODE     SALARY
--------- -------------------- -------- ---------- ----------

30020013  GARRET               UKL            4200          0
                               UKL            4150          0
                                                 0          0

30016112  TAILOR               UKL            7450          0
                               UKL            7350          0
                               UKL            6700          0

20017600  PIETSCH              USD           22000          0
                               USD           20200          0
                               USD           18700          0
```

Da bei Verwendung eines `WRITE`-Statements die Werte nebeneinander (statt untereinander) ausgegeben werden, kann dies eventuell einen (möglicherweise unerwünschten) automatischen Zeilenvorschub auslösen.

Wenn Sie statt einer ganzen Periodengruppe nur ein Feld (z.B. `SALARY`) aus einer Periodengruppe benutzen und, wie im folgenden Beispiel zwischen `NAME` und `JOB-TITLE`, zusätzlich einen Zeilenvorschub, d.h. einen Schrägstrich (/), einfügen, wird der Report übersichtlicher:

Beispiel für Index-Notation im WRITE-Statement

```
** Example 'WRITEX03': WRITE (with index notation)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
```

```
  WRITE PERSONNEL-ID NAME / JOB-TITLE SALARY (1:3)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX03`:

```
Page      1                                                  04-11-11  14:15:55

30020013 GARRET
TYPIST                          4200       4150          0

30016112 TAILOR
WAREHOUSEMAN                    7450       7350       6700

20017600 PIETSCH
SECRETARY                      22000      20200      18700
```

37 Seitenüberschriften, Seitenvorschübe und Leerzeilen

- Standard-Seitenüberschrift ... 318
- Seitenüberschrift unterdrücken — die NOTITLE-Option ... 318
- Eigene Seitenüberschrift definieren — das WRITE TITLE-Statement ... 319
- Logische Seite und physische Seite ... 323
- Seitenlänge — der PS-Parameter ... 325
- Seitenvorschub ... 325
- Neue Seite mit Titel ... 328
- Seiten-Fußzeile — das WRITE TRAILER-Statement ... 329
- Leerzeilen erzeugen — das SKIP-Statement ... 331
- AT TOP OF PAGE-Statement ... 333
- AT END OF PAGE-Statement ... 334
- Weiteres Beispiel ... 335

Dieses Kapitel beschreibt verschiedene Möglichkeiten, wie Sie den Seitenumbruch in einem Report sowie die Ausgabe von Seitenüberschriften am Anfang jeder Seite des Reports und die Erzeugung von Leerzeilen in einem Ausgabe-Report beeinflussen können.

Standard-Seitenüberschrift

Natural generiert für jede über ein `DISPLAY`- oder `WRITE`-Statement erzeugte Ausgabeseite automatisch eine Standard-Kopfzeile. Diese Kopfzeile enthält die Seitennummer sowie Datum und Uhrzeit.

Beispiel:

```
WRITE 'HELLO'
END
```

Das obige Programm erzeugt folgende Ausgabe mit einer Standard-Kopfzeile:

```
Page      1                                                04-12-14  13:19:33

HELLO
```

Seitenüberschrift unterdrücken — die NOTITLE-Option

Falls Sie Ihren Report ohne Kopfzeile ausgeben möchten, geben Sie im `DISPLAY`- bzw. `WRITE`-Statement das Schlüsselwort `NOTITLE` an.

Beispiel - DISPLAY mit NOTITLE:

```
** Example 'DISPLX20': DISPLAY (with NOTITLE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (5) EMPLOY-VIEW BY CITY FROM 'BOSTON'
  DISPLAY NOTITLE NAME FIRST-NAME CITY
END-READ
END
```

Ausgabe des Programms `DISPLX20`:

```
        NAME                FIRST-NAME               CITY
-------------------- -------------------- --------------------

SHAW                 LESLIE               BOSTON
STANWOOD             VERNON               BOSTON
CREMER               WALT                 BOSTON
PERREAULT            BRENDA               BOSTON
COHEN                JOHN                 BOSTON
```

Beispiel - WRITE mit NOTITLE:

```
WRITE NOTITLE 'HELLO'
END
```

Das obige Programm erzeugt folgende Ausgabe ohne Kopfzeile:

```
HELLO
```

Eigene Seitenüberschrift definieren — das WRITE TITLE-Statement

Wenn Sie statt der Natural-Standard-Kopfzeile eine eigene Kopfzeile ausgeben möchten, verwenden Sie dazu das Statement `WRITE TITLE`.

In diesem Abschnitt werden folgende Themen behandelt:

- Text für Ihre Überschrift angeben
- Leerzeilen nach der Überschrift angeben
- Überschriften-Ausrichtung und/oder -Unterstreichung
- Überschrift mit Seitenzahl

Text für Ihre Überschrift angeben

Mit dem Statement WRITE TITLE geben Sie (in Apostrophen) den Text Ihrer Kopfzeile an.

```
WRITE TITLE 'THIS IS MY PAGE TITLE'
WRITE 'HELLO'
END
```

Mit dem obigen Programm wird die folgende Ausgabe erzeugt:

```
                             THIS IS MY PAGE TITLE
HELLO
```

Leerzeilen nach der Überschrift angeben

Mit der SKIP-Option des WRITE TITLE-Statements können Sie bestimmen, wieviele Leerzeilen unter der Kopfzeile ausgegeben werden sollen. Nach dem Schlüsselwort SKIP geben Sie die Anzahl der gewünschten Leerzeilen an:

```
WRITE TITLE 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

Mit dem obigen Programm wird die folgende Ausgabe erzeugt:

```
                             THIS IS MY PAGE TITLE


HELLO
```

SKIP kann nicht nur in einem WRITE TITLE-Statement, sondern auch als eigenständiges Statement verwendet werden.

Überschriften-Ausrichtung und/oder -Unterstreichung

Standardmäßig wird die Kopfzeile zentriert und ohne Unterstreichung ausgegeben.

Das `WRITE TITLE`-Statement bietet Ihnen aber auch die Möglichkeit, eine Seitenüberschrift linksbündig (`LEFT JUSTIFIED`) und/oder unterstrichen (`UNDERLINED`) auszugeben.

Option	Auswirkung
`LEFT JUSTIFIED`	Bewirkt, dass die Überschrift linksbündig angezeigt wird.
`UNDERLINED`	Bewirkt, dass die Überschrift unterstrichen angezeigt wird. Das Unterstreichen legt die Breite der Zeile fest (siehe auch Natural Profil- und Session-Parameter `LS`). Standardmäßig werden Überschriften mit einem Bindestrich (-) unterstrichen. Mit dem Session-Parameter `UC` können Sie aber ein anderes Zeichen angeben, das als Zeichen zum Unterstreichen benutzt werden soll (siehe ***Unterstreichungszeichen für Überschriften***).

Das folgende Beispiel zeigt die Auswirkungen der Optionen `LEFT JUSTIFIED` und `UNDERLINED`:

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'THIS IS MY PAGE TITLE'
SKIP 2
WRITE 'HELLO'
END
```

Mit dem obigen Programm wird die folgende Ausgabe erzeugt:

```
THIS IS MY PAGE TITLE
------------------------------------------------------------------------------


HELLO
```

Das `WRITE TITLE`-Statement wird jedesmal ausgeführt, wenn eine neue Reportseite initiiert wird.

Überschrift mit Seitenzahl

In den folgenden Beispielen wird die Systemvariable *PAGE-NUMBER in Verbindung mit dem Statement WRITE TITLE zur Ausgabe der Seitenzahl in der Überschriftenzeile benutzt.

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

Ausgabe des Programms WTITLX01:

```
YEAR:  1980               PAGE      1
YEAR          MAKE         MAINT-COST
----- -------------------- ----------

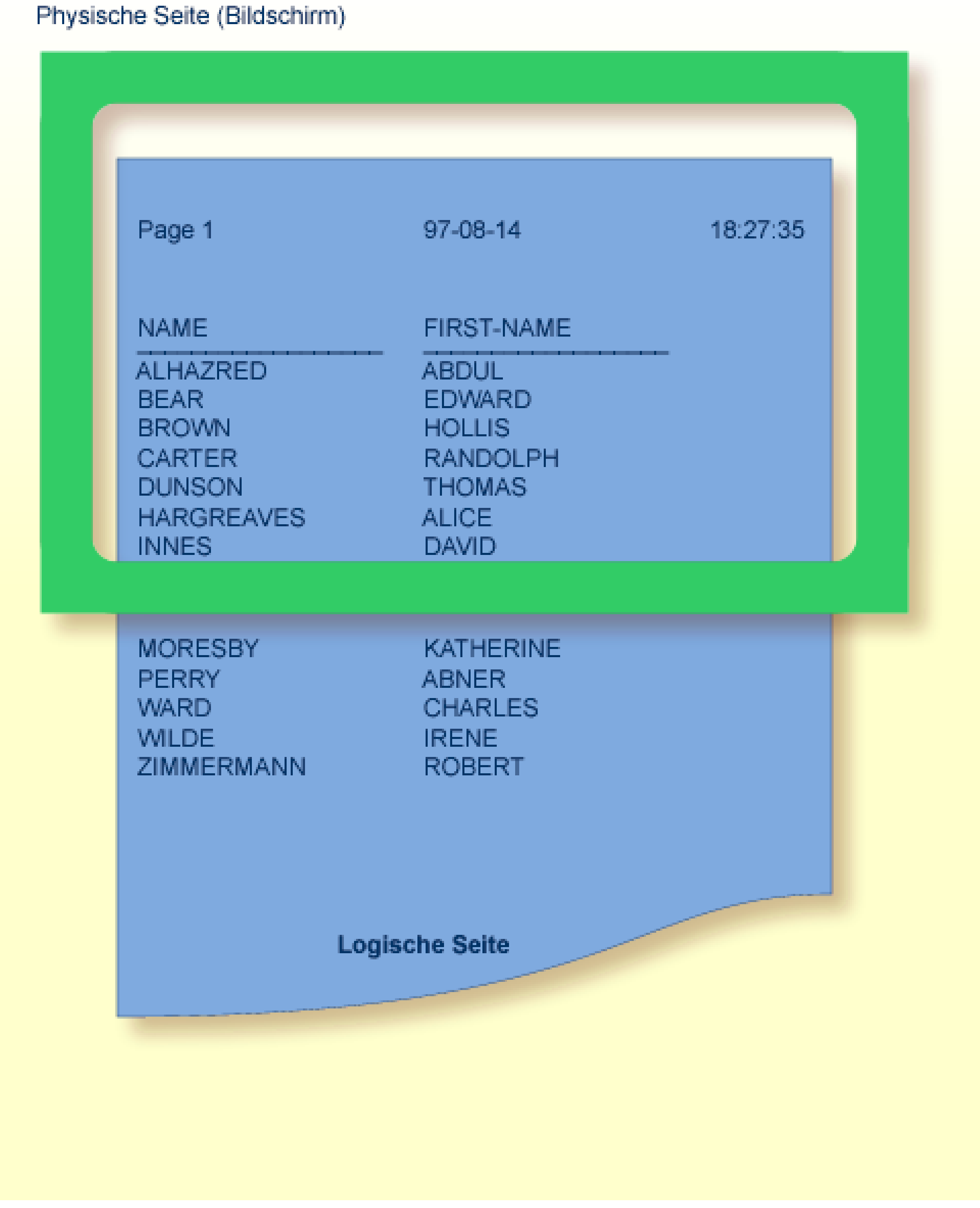
 1980 RENAULT                   20000
 1980 RENAULT                   20000
 1980 PEUGEOT                   20000
```

Logische Seite und physische Seite

Eine *logische Seite* ist die von einem Natural-Programm erzeugte Ausgabe.

Eine *physische Seite* ist Ihr Bildschirm, auf dem die Ausgabe angezeigt wird; oder es ist das Stück Papier, auf dem die Ausgabe ausgedruckt wird.

Falls mehr Zeilen ausgegeben werden als auf einen Bildschirm passen, ist die logische Seite länger als die physische Seite, und die restlichen Zeilen werden auf dem nächsten Schirm angezeigt.



Anmerkung: Falls Informationen, die Sie unten auf dem Schirm anzeigen möchten (z.B. mit einem `WRITE TRAILER`- oder `AT END OF PAGE`-Statement erzeugte Ausgaben), erst auf dem nächsten Schirm ausgegeben werden, verkleinern Sie die logische Seitenlänge entsprechend (mit dem Session-Parameter `PS`, wie unten beschrieben).

Seitenlänge — der PS-Parameter

Mit dem Parameter PS (Page Size for Natural Reports) bestimmen Sie die maximale Anzahl der Zeilen einer (logischen) Ausgabeseite.

Wenn die Anzahl der mit dem PS-Parameter angegebenen Zeilen erreicht ist, dann erfolgt ein Seitenvorschub (es sei denn, der Seitenvorschub wird über ein NEWPAGE- oder ein EJECT-Statement gesteuert; siehe unten).

Der PS-Parameter kann entweder auf Session-Ebene mit dem Systemkommando GLOBALS gesetzt werden oder innerhalb eines Programms mit den folgenden Statements:

Auf Report-Ebene:

- FORMAT PS=*nn*

Auf Statement-Ebene:

- DISPLAY (PS=*nn*)
- WRITE (PS=*nn*)
- WRITE TITLE (PS=*nn*)
- WRITE TRAILER (PS=*nn*)
- INPUT (PS=*nn*)

Seitenvorschub

Ein Seitenvorschub kann durch eine der folgenden Methoden erreicht werden:

- Seitenvorschub durch EJ-Parameter
- Seitenvorschub durch EJECT oder NEWPAGE-Statement
- EJECT/NEWPAGE bei weniger als n restlichen Zeilen auf der Seite

Diese Methoden werden im Folgenden erörtert.

Seitenvorschub durch EJ-Parameter

Mit dem Session-Parameter `EJ` (Page Eject) bestimmen Sie, ob Seitenvorschübe ausgeführt werden sollen oder nicht. Standardmäßig gilt `EJ=ON`, d.h. Seitenvorschübe werden wie angegeben ausgeführt.

Wenn Sie `EJ=OFF` angeben, werden Seitenvorschub-Informationen ignoriert. Dies kann bei Testläufen, bei denen Seitenumbrüche keine Rolle spielen, sinnvoll sein, um Papier zu sparen.

Der Seitenvorschub-Parameter `EJ` kann auf Session-Ebene mit dem Systemkommando `GLOBALS` gesetzt werden:

```
GLOBALS EJ=OFF
```

Die Einstellung des `EJ`-Parameters wird durch das `EJECT`-Statement überschrieben.

Seitenvorschub durch EJECT oder NEWPAGE-Statement

Folgende Themen werden behandelt:

- Seitenvorschub ohne Überschrift/Fußzeile auf der nächsten Seite
- Seitenvorschub mit Verarbeitung am Ende/Anfang der Seite

Seitenvorschub ohne Überschrift/Fußzeile auf der nächsten Seite

Das `EJECT`-Statement bewirkt einen Seitenvorschub, *ohne* dass auf der neuen Seite eine Kopfzeile oder Standard-Seitenüberschrift generiert wird. An Seitenanfang und Seitenende gebundene Verarbeitungen wie `WRITE TRAILER` oder `AT END OF PAGE`, `WRITE TITLE`, `AT TOP OF PAGE` oder `*PAGE-NUMBER` werden nicht ausgeführt.

Das `EJECT`-Statement hat Priorität vor dem `EJ`-Parameter.

Seitenvorschub mit Verarbeitung am Ende/Anfang der Seite

Das `NEWPAGE`-Statement hingegen bewirkt einen Seitenvorschub *mit* Ausführung der für Seitenanfang und Seitenende festgelegten Verarbeitungen. Eine Fußzeile wird ausgegeben, falls spezifiziert; eine standardmäßige oder benutzerdefinierte Kopfzeile wird auf der neuen Seite ausgegeben (es sei denn, das betreffende `DISPLAY`- bzw. `WRITE`-Statement enthält die Option `NOTITLE`).

Wird kein `NEWPAGE`-Statement verwendet, so ergibt sich der Seitenvorschub aus der mit dem Parameter PS definierten Seitenlänge (siehe ***Seitenlänge - der PS-Parameter*** oben).

EJECT/NEWPAGE bei weniger als n restlichen Zeilen auf der Seite

Das `NEWPAGE`- wie das `EJECT`-Statement erlauben es, eine `WHEN LESS THAN n LINES LEFT`-Klausel anzugeben. Mit dieser Klausel geben Sie eine Zeilenanzahl *n* an; das `NEWPAGE`- bzw. `EJECT`-Statement wird dann nur ausgeführt, wenn zum Zeitpunkt der Verarbeitung des Statements weniger als *n* Zeilen auf der aktuellen Seite zur Verfügung stehen.

Beispiel 1:

```
FORMAT PS=55
...
NEWPAGE WHEN LESS THAN 7 LINES LEFT
...
```

In diesem Beispiel ist die Seitenlänge mit 55 Zeilen angegeben.

Sind zu dem Zeitpunkt, zu dem das `NEWPAGE`-Statement verarbeitet wird, auf der aktuellen Seite nur noch 6 oder weniger Zeilen übrig, wird das `NEWPAGE`-Statement ausgeführt. Sind 7 oder mehr übrig, wird es nicht ausgeführt, und der Seitenvorschub erfolgt in Abhängigkeit vom `PS`-Parameter, also nach 55 Zeilen.

Beispiel 2:

```
** Example 'NEWPAX02': NEWPAGE (in combination with EJECT and
**                     parameter PS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FORMAT PS=15
*
READ (9) EMPLOY-VIEW BY CITY STARTING FROM 'BOSTON'
  AT START OF DATA
    EJECT
    WRITE /// 20T '%' (29) /
              20T '%%'                         47T '%%' /
              20T '%%' 3X 'REPORT OF EMPLOYEES' 47T '%%' /
              20T '%%' 3X '  SORTED BY CITY   ' 47T '%%' /
              20T '%%'                         47T '%%' /
              20T '%' (29) /
    NEWPAGE
  END-START
  AT BREAK OF CITY
    NEWPAGE WHEN LESS 3 LINES LEFT
```

```
  END-BREAK
  DISPLAY CITY (IS=ON) NAME JOB-TITLE
END-READ
END
```

Neue Seite mit Titel

Das NEWPAGE-Statement bietet darüber hinaus eine WITH TITLE-Option. Ohne diese Option wird entweder die Standard-Kopfzeile ausgegeben oder ein WRITE TITLE-Statement bzw. eine NOTITLE-Option ausgeführt.

Mit der WITH TITLE-Option können Sie für einen mit NEWPAGE ausgelösten Seitenvorschub eine eigene Kopfzeile ausgeben, die dann Priorität vor allen anderen Seitenüberschrift-Anweisungen hat. Die Syntax der WITH TITLE-Klausel entspricht der des WRITE TITLE-Statements.

Beispiel:

```
NEWPAGE WITH TITLE LEFT JUSTIFIED 'PEOPLE LIVING IN BOSTON:'
```

Das folgende Beispielprogramm zeigt die Verwendung des PS-Parameters und des NEWPAGE-Statements. Außerdem wird hier die Natural-Systemvariable *PAGE-NUMBER verwendet, die jeweils die aktuelle Seitenzahl enthält.

```
** Example 'NEWPAX01': NEWPAGE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 DEPT
END-DEFINE
*
FORMAT PS=20
READ (5) VIEWEMP BY CITY STARTING FROM 'M'
  DISPLAY NAME 'DEPT' DEPT 'LOCATION' CITY
  AT BREAK OF CITY
    NEWPAGE WITH TITLE LEFT JUSTIFIED
           'EMPLOYEES BY CITY - PAGE:' *PAGE-NUMBER
  END-BREAK
END-READ
END
```

Ausgabe des Programms NEWPAX01:

Beachten Sie, wann der Seitenvorschub erfolgt, sowie die Kopfzeile der neuen Seite:

```
Page      1                                                   04-11-11  14:15:54

        NAME          DEPT        LOCATION
-------------------- ------ --------------------

FICKEN               TECH10 MADISON
KELLOGG              TECH10 MADISON
ALEXANDER            SALE20 MADISON
```

Seite 2:

```
EMPLOYEES BY CITY - PAGE:      2
        NAME          DEPT        LOCATION
-------------------- ------ --------------------

DE JUAN              SALE03 MADRID
DE LA MADRID         PROD01 MADRID
```

Seite 3:

```
EMPLOYEES BY CITY - PAGE: 3
```

Seiten-Fußzeile — das WRITE TRAILER-Statement

Folgende Themen werden behandelt:

- Seiten-Fußzeile angeben
- Logische Seitenlänge berücksichtigen
- Ausrichtung und/oder Unterstreichung der Seiten-Fußzeile

Seiten-Fußzeile angeben

Mit dem Statement `WRITE TRAILER` können Sie einen Text in Apostrophen (') angeben, der als Fußzeile am Ende jeder Seite ausgegeben werden soll.

```
WRITE TRAILER 'THIS IS THE END OF THE PAGE'
```

Das Statement wird ausgeführt vor einem `SKIP`- oder `NEWPAGE`-Statement oder am Ende einer logischen Seite.

Logische Seitenlänge berücksichtigen

Die Prüfung, ob das Ende einer logischen Seite erreicht ist, erfolgt erst, *nachdem* ein `WRITE`- oder `DISPLAY`-Statement vollständig ausgeführt ist. Daher kann es vorkommen, dass der Umfang einer logischen Seite (d.h. die Anzahl der mit einem `DISPLAY`- bzw. `WRITE`-Statement ausgegebenen Zeilen) eine physische Seite überschreitet, bevor das `WRITE TRAILER`-Statement ausgeführt wird.

Um sicherzustellen, dass die Fußzeilen jeweils am Ende einer physischen Seite erscheinen, sollten Sie die logische Seitenlänge (Session-Parameter `PS`) so festlegen, dass sie entsprechend kleiner als die physische Seitenlänge ist.

Ausrichtung und/oder Unterstreichung der Seiten-Fußzeile

Standardmäßig wird die Seiten-Fußzeile zentriert auf der Seite und nicht unterstrichen ausgegeben.

Das `WRITE TRAILER`-Statement bietet Ihnen aber auch die Möglichkeit, eine Fußzeile linksbündig (`LEFT JUSTIFIED`) und/oder unterstrichen (`UNDERLINED`) auszugeben:

Option	Auswirkung
`LEFT JUSTIFIED`	Bewirkt, dass die Fußzeile linksbündig angezeigt wird.
`UNDERLINED`	Bewirkt, dass die Fußzeile unterstrichen angezeigt wird. Das Unterstreichen erfolgt über die ganze Breite der Zeile fest (siehe auch Natural Profil- und Session-Parameter `LS`). Standardmäßig werden Überschriften mit einem Bindestrich (–) unterstrichen. Mit dem Session-Parameter `UC` können Sie aber ein anderes Zeichen angeben, das als Zeichen zum Unterstreichen benutzt werden soll (siehe ***Unterstreichungszeichen für Überschriften***).

Die folgenden Beispiele zeigen die Verwendung der Optionen `LEFT JUSTIFIED` und `UNDERLINED` des `WRITE TRAILER`-Statements:

Beispiel 1:

```
WRITE TRAILER LEFT JUSTIFIED UNDERLINED 'THIS IS THE END OF THE PAGE'
```

Beispiel 2:

```
** Example 'WTITLX02': WRITE TITLE AND WRITE TRAILER
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Leerzeilen erzeugen — das SKIP-Statement

Das `SKIP`-Statement wird zum Erzeugen von einer oder mehrerer Leerzeilen in einem Ausgabe-Report benutzt.

Beispiel 1 - SKIP in Verbindung mit WRITE und DISPLAY:

```
** Example 'SKIPX01': SKIP (in conjunction with WRITE and DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
     'PEOPLE LIVING IN SALT LAKE CITY AS OF' *DAT4E 7X
     'PAGE:' *PAGE-NUMBER
SKIP 3
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY NAME / FIRST-NAME CITY ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Beispiel 2 - SKIP in Verbindung mit DISPLAY VERT:

```
** Example 'SKIPX02': SKIP (in conjunction with DISPLAY VERT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
END-DEFINE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE VERT
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
*
NEWPAGE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
END
```

AT TOP OF PAGE-Statement

Mit dem Statement AT TOP OF PAGE können Sie eine beliebige Verarbeitung angeben, die jedesmal ausgeführt werden soll, wenn eine neue Reportseite beginnt.

Erzeugt das AT TOP OF PAGE-Statement eine Ausgabe, so wird diese unterhalb der Seiten- Kopfzeile (mit einer Leerzeile dazwischen) ausgegeben.

Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

Beispiel:

```
** Example 'ATTOPX01': AT TOP OF PAGE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 MAR-STAT
  2 BIRTH
  2 CITY
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE (AL=10)
          NAME DEPT JOB-TITLE CITY 5X
          MAR-STAT 'DATE OF/BIRTH' BIRTH (EM=YY-MM-DD)
  /*
  AT TOP OF PAGE
    WRITE /    '-BUSINESS INFORMATION-'
          26X '-PRIVATE INFORMATION-'
  END-TOPPAGE
END-READ
END
```

Ausgabe des Programms ATTOPX01:

```
-BUSINESS INFORMATION-                              -PRIVATE INFORMATION-
   NAME    DEPARTMENT  CURRENT      CITY             MARITAL   DATE OF
              CODE     POSITION                       STATUS    BIRTH
---------- ---------- ---------- ----------         ---------- --------

CREMER     TECH10     ANALYST    GREENVILLE         S          70-01-01
MARKUSH    SALE00     TRAINEE    LOS ANGELE         D          79-03-14
GEE        TECH05     MANAGER    CHAPEL HIL         M          41-02-04
KUNEY      TECH10     DBA        DETROIT            S          40-02-13
NEEDHAM    TECH10     PROGRAMMER CHATTANOOG         S          55-08-05
JACKSON    TECH10     PROGRAMMER ST LOUIS           D          70-01-01
PIETSCH    MGMT10     SECRETARY  VISTA              M          40-01-09
PAUL       MGMT10     SECRETARY  NORFOLK            S          43-07-07
HERZOG     TECH05     MANAGER    CHATTANOOG         S          52-09-16
DEKKER     TECH10     DBA        MOBILE             W          40-03-03
```

AT END OF PAGE-Statement

Mit dem Statement `AT END OF PAGE` können Sie eine beliebige Verarbeitung angeben, die jedesmal ausgeführt werden soll, wenn das Ende einer Reportseite erreicht wird.

Erzeugt das `AT END OF PAGE`-Statement eine Ausgabe, so wird diese unterhalb der (mit dem `WRITE TRAILER`-Statement angegebenen) **Seiten-Fußzeile** ausgegeben.

Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

Dieselben Anmerkungen bezüglich logischer und physischer Seitenlängen, die für das `DISPLAY`- und `WRITE`-Statement gelten (vgl. **oben**), treffen auch auf das `AT END OF PAGE`-Statement zu.

Beispiel:

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
```

```
  /*
  AT END OF PAGE
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

Ausgabe des Programms `ATENPX01`:

```
        NAME                      CURRENT            SALARY
                                 POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                        34000
MARKUSH              TRAINEE                        22000
GEE                  MANAGER                        39500
KUNEY                DBA                            40200
NEEDHAM              PROGRAMMER                     32500
JACKSON              PROGRAMMER                     33000
PIETSCH              SECRETARY                      22000
PAUL                 SECRETARY                      23000
HERZOG               MANAGER                        48500
DEKKER               DBA                            48000


AVERAGE SALARY: ...      34270
```

Weiteres Beispiel

Siehe das folgende Beispielprogramm:

- ***DISPLX21 - DISPLAY-Statement (mit Schrägstrich '/' und zum Vergleich mit WRITE)***

38 Spaltenüberschriften

- Standard-Spaltenüberschriften 338
- Standard-Spaltenüberschriften unterdrücken — die NOHDR-Option 339
- Eigene Spaltenüberschriften definieren 339
- NOTITLE und NOHDR kombinieren 340
- Spaltenüberschriften zentrieren — der HC-Parameter 340
- Breite von Spaltenüberschriften — der HW-Parameter 341
- Füllzeichen für Überschriften — die Parameter FC und GC 341
- Unterstreichungszeichen für Überschriften — der UC-Parameter 342
- Spaltenüberschriften unterdrücken — die Schrägstrich-Notation ('/') 344
- Weitere Beispiele für Spaltenüberschriften 345

Dieses Kapitel beschreibt verschiedene Möglichkeiten, wie Sie die Anzeige der von einem DISPLAY-Statement erzeugten Spaltenüberschriften beeinflussen können.

Standard-Spaltenüberschriften

Standardmäßig wird jedes mit einem DISPLAY-Statement ausgegebene Datenbankfeld mit einer (für das Feld im DDM definierten) Standard-Spaltenüberschrift ausgegeben.

```
** Example 'DISPLX01': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END
```

Ausgabe des Programms DISPLX01:

Das obige Beispielprogramm verwendet Standard-Spaltenüberschriften und erzeugt folgende Ausgabe:

```
Page      1                                                04-11-11  14:15:54

PERSONNEL         NAME                  CURRENT
   ID                                  POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Standard-Spaltenüberschriften unterdrücken — die NOHDR-Option

Wünschen Sie in Ihrem Report keine Spaltenüberschriften, geben Sie im DISPLAY-Statement das Schlüsselwort NOHDR an, zum Beispiel:

```
DISPLAY NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Eigene Spaltenüberschriften definieren

Wenn Sie statt der Standard-Spaltenüberschriften eigene Spaltenüberschriften ausgeben möchten, geben Sie unmittelbar vor dem jeweiligen Feld einen Text in Apostrophen (') an, wobei '*text*' die für das Feld zu verwendende Spaltenüberschrift ist.

```
** Example 'DISPLX08': DISPLAY  (with column title in 'text')
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID
          'EMPLOYEE' NAME
          'POSITION' JOB-TITLE
END-READ
END
```

Ausgabe des Programms DISPLX08:

Das obige Programm enthält für das Feld NAME die Spaltenüberschrift EMPLOYEE und für das Feld JOB-TITLE die Spaltenüberschrift POSITION; für das Feld PERSONNEL-ID wird die Standard-Spaltenüberschrift verwendet. Das Programm erzeugt folgende Ausgabe:

```
Page      1                                            04-11-11  14:15:54

PERSONNEL       EMPLOYEE               POSITION
   ID
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

NOTITLE und NOHDR kombinieren

Zur Ausgabe eines Reports ohne Kopfzeilen und Spaltenüberschriften geben Sie die Optionen NOTITLE und NOHDR gleichzeitig an, und zwar in der folgenden Reihenfolge:

```
DISPLAY NOTITLE NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Spaltenüberschriften zentrieren — der HC-Parameter

Standardmäßig werden Spaltenüberschriften zentriert über den Spalten ausgegeben. Mit dem Session-Parameter HC (Header Centering) können Sie die Ausrichtung der Spaltenüberschriften beeinflussen:

HC=L	Spaltenüberschriften werden linksbündig ausgerichtet.
HC=R	Spaltenüberschriften werden rechtsbündig ausgerichtet.
HC=C	Spaltenüberschriften werden zentriert. Dies ist die Standardeinstellung.

Sie können den HC-Parameter in einem FORMAT-Statement angeben; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Elementebene.

Beispiel für die Angabe des HC-Parameters auf Statement-Ebene, d.h. für die linksbündige Ausrichtung *aller* Spaltenüberschriften:

```
DISPLAY (HC=L) PERSONNEL-ID NAME JOB-TITLE
```

Breite von Spaltenüberschriften — der HW-Parameter

Mit dem Session-Parameter `HW` (Header Width) bestimmen Sie die Breite einer von einem `DISPLAY`-Statement erzeugten Spalte.

`HW=ON`	Die Breite einer `DISPLAY`-Spalte wird entweder durch die Länge des Feldes oder durch die Länge der Spaltenüberschrift bestimmt, je nachdem was länger ist. Dies ist die Standardeinstellung.
`HW=OFF`	Wenn Sie `HW=OFF` angeben, wird die Breite einer `DISPLAY`-Spalte allein durch die Länge des Feldes bestimmt. Bitte beachten Sie, dass `HW=OFF` nur bei `DISPLAY`-Statements, die keine Spaltenüberschriften erzeugen, wirkt; d.h. bei einem ersten `DISPLAY`-Statement mit `NOHDR`-Option, oder bei einem nachfolgenden `DISPLAY`-Statement.

Sie können den `HW`-Parameter in einem `FORMAT`-Statement verwenden; er gilt dann für den gesamten Report. Sie können ihn auch in einem `DISPLAY`-Statement angeben, und zwar sowohl auf Statement- wie auf Elementebene.

Füllzeichen für Überschriften — die Parameter FC und GC

Mit dem Session-Parameter `FC` (Filler Character) bestimmen Sie das *Füllzeichen*, das auf beiden Seiten der von einem `DISPLAY`-Statement erzeugten *Überschrift* über die gesamte Breite der Spalte erscheint. Voraussetzung ist, dass die Spaltenbreite durch die Feldlänge und nicht durch die Überschrift bestimmt wird (vgl. `HW`-Parameter und Beschreibung **oben**), sonst hat der `FC`-Parameter keine Wirkung.

Wenn eine Feldgruppe oder eine Periodengruppe mit einem `DISPLAY`-Statement ausgegeben wird, wird eine *Gruppenüberschrift* über den Überschriften der einzelnen Felder der Gruppe ausgegeben. Mit dem Session-Parameter `GC` (Group Filler Character) bestimmen Sie das *Füllzeichen*, das auf beiden Seiten der Gruppenüberschrift erscheinen soll.

Während der `FC`-Parameter für Überschriften einzelner Felder gilt, bezieht sich der `GC`-Parameter auf Überschriften für Feldgruppen.

Sie können die Parameter FC und GC in einem FORMAT-Statement verwenden; sie gelten dann für den gesamten Report. Sie können sie auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Elementebene.

```
** Example 'FORMAX01': FORMAT (with parameters FC, GC)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 INCOME (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
FORMAT FC=* GC=$
*
READ (3) VIEWEMP BY NAME
  DISPLAY NAME (FC==) INCOME (1)
END-READ
END
```

Ausgabe des Programms FORMAX01:

```
Page      1                                                04-11-11  14:15:54

========NAME======== $$$$$$$$$$$$INCOME$$$$$$$$$$$$

                     CURRENCY **ANNUAL** **BONUS***
                         CODE     SALARY
-------------------- -------- ---------- ----------

ABELLAN              PTA         1450000          0
ACHIESON             UKL           10500          0
ADAM                 FRA          159980      23000
```

Unterstreichungszeichen für Überschriften — der UC-Parameter

Standardmäßig werden Kopfzeilen und Überschriften mit einem Bindestrich (-) unterstrichen.

Mit dem Session-Parameter UC (Underlining Character) können Sie ein anderes Zeichen bestimmen, das als Unterstreichungszeichen verwendet werden soll.

Sie können den UC-Parameter in einem FORMAT-Statement verwenden; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Elementebene.

```
** Example 'FORMAX02': FORMAT (with parameter UC)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
FORMAT UC==
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'EMPLOYEES REPORT'
SKIP 1
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID (UC=*) NAME JOB-TITLE
END-READ
END
```

Im obigen Programm ist der UC-Parameter auf Programmebene und auf Feldebene gesetzt: das im FORMAT-Statement angegebene Unterstreichungszeichen (=) gilt für den ganzen Report außer für das Feld PERSONNEL-ID, für das ein anderes Unterstreichungszeichen (*) angegeben ist.

Ausgabe des Programms FORMAX02:

```
EMPLOYEES REPORT
===============================================================================

PERSONNEL         NAME                  CURRENT
   ID                                  POSITION
********* ==================== =========================

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Spaltenüberschriften unterdrücken — die Schrägstrich-Notation ('/')

Mit der Notation Apostroph-Schrägstrich-Apostroph (`'/'`) können Sie die Ausgabe von Standard-Spaltenüberschriften für einzelne Felder in einem `DISPLAY`-Statement unterdrücken.

Im Gegensatz zur `NOHDR`-Option, mit der Sie die Ausgabe von Standard-Spaltenüberschriften für *alle* Spalten unterdrücken können, können Sie mit der `'/'`-Notation die Überschrift für *eine einzelne* Spalte unterdrücken.

Dazu geben Sie die Notation `'/'` in einem `DISPLAY`-Statement jeweils unmittelbar vor dem Namen des Feldes an, dessen Spaltenüberschrift unterdrückt werden soll.

Zwei Beispiele zum Vergleich:

Beispiel 1:

```
DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

In diesem Beispiel werden die Standardüberschriften aller drei Spalten ausgegeben:

```
Page      1                                                04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                          ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             20011000  SALES PERSON
```

Beispiel 2:

```
DISPLAY '/' NAME PERSONNEL-ID JOB-TITLE
```

In diesem Beispiel wird mit der Notation `'/'` die Spaltenüberschrift für das Feld `NAME` unterdrückt:

```
Page      1                                               04-11-11  14:15:54

                     PERSONNEL          CURRENT
                        ID             POSITION
                     --------- -------------------------


ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             20011000  SALES PERSON
```

Weitere Beispiele für Spaltenüberschriften

Siehe die folgenden Beispiel-Programme:

- *DISPLX15 – DISPLAY-Statement (mit FC, UC)*
- *DISPLX16 – DISPLAY-Statement (mit '/', 'text', 'text/text')*

39 Parameter zur Beeinflussung der Ausgabe von Feldern

- Übersicht über Feldausgabe-relevante Parameter ... 348
- Vorangestellte Zeichen — der LC-Parameter ... 348
- Vorangestellte Zeichen im Unicode-Format — der LCU Parameter ... 349
- Einfügungszeichen — der IC-Parameter ... 349
- Einfügungszeichen im Unicode-Format — der ICU Parameter ... 350
- Nachgestellte Zeichen — der TC-Parameter ... 350
- Vorangestellte Zeichen im Unicode-Format — der TCU Parameter ... 350
- Ausgabelänge — der AL- und der NL-Parameter ... 351
- Ausgabelänge — der DL Parameter ... 351
- Vorzeichen-Stelle — der SG-Parameter ... 353
- Ausgabe identischer Werte unterdrücken — der IS-Parameter ... 355
- Nullwerte anzeigen — der ZP-Parameter ... 357
- Leerzeilen unterdrücken — der ES-Parameter ... 357
- Weitere Beispiele für Feldausgabe-relevante Parameter ... 359

Dieses Kapitel erörtert die Benutzung der Natural Profil- und/oder Session-Parameter, die Sie zum Steuern des Ausgabe-Formats von Feldern verwenden können.

Übersicht über Feldausgabe-relevante Parameter

Natural bietet eine Reihe von Profil- und/oder Session-Parametern, mit denen Sie die Art, in der Felder ausgegeben werden, beeinflussen können:

Parameter	Funktion
LC, IC und TC	Mit diesen Session-Parametern können Sie Zeichen angeben, die vor bzw. nach einem Feld bzw. vor einem Feldwert angezeigt werden sollen.
LCU, ICU und TCU	Mit diesen Session-Parametern können Sie Zeichen im Unicode-Format angeben, die vor bzw. nach einem Feldwert angezeigt werden sollen.
AL und NL	Mit diesen Session-Parametern können Sie die Ausgabelänge von Feldern vergrößern oder verkleinern.
DL	Mit diesem Session-Parameter können Sie die Standard-Ausgabelänge für ein alphanumerisches Maskenfeld des Formats U angeben.
SG	Mit diesem Session-Parameter können Sie bestimmen, ob negative Werte mit oder ohne Minuszeichen angezeigt werden sollen.
IS	Mit diesem Session-Parameter können Sie die Anzeige von Feldwerten, die mit dem jeweils vorigen Feldwert identisch sind, unterdrücken.
ZP	Mit diesem Profil- und Session-Parameter können Sie bestimmen, ob Feldwerte, die `0` sind, angezeigt werden sollen oder nicht.
ES	Mit diesem Session-Parameter können Sie die Anzeige von Leerzeilen, die von einem `DISPLAY`- oder `WRITE`-Statement erzeugt werden, unterdrücken.

Diese Parameter werden im Folgenden behandelt.

Vorangestellte Zeichen — der LC-Parameter

Mit dem Session-Parameter `LC` (Leading Characters) geben Sie an, welche Zeichen unmittelbar *vor einem Feld* ausgegeben werden, das von einem `DISPLAY`-Statement ausgegeben wird. Die Breite der Ausgabespalte wird entsprechend vergrößert. Sie können 1 bis 10 Zeichen angeben.

Standardmäßig sind die Werte in alphanumerischen Feldern linksbündig ausgerichtet und die Werte in numerischen Feldern rechtsbündig. (Mit dem Session-Parameter `AD` können Sie diese Ausrichtung ändern. Bei einem alphanumerischen Feld erscheint ein vorangestelltes Zeichen daher unmittelbar vor dem Feldwert; bei einem numerischen Feld kann es dagegen vorkommen, dass zwischen dem `LC`-Zeichen und dem Feldwert Leerstellen bleiben.

Der LC-Parameter kann mit den folgenden Statements verwendet werden:

- FORMAT
- DISPLAY

Er kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Vorangestellte Zeichen im Unicode-Format — der LCU Parameter

Der Session-Parameter LCU ist identisch mit dem Session-Parameter LC. Der Unterschied ist, dass alle vorangestellten Zeichen immer im Unicode-Format gespeichert werden.

Das gibt Ihnen die Möglichkeit, vorangestellte Zeichen mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Weitere Informationen siehe *Unicode and Code Page Support in the Natural Programming Language, Session Parameters*, Abschnitt *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

Die Parameter LCU und ICU dürfen nicht beide gleichzeitig für ein Feld angegeben werden.

Einfügungszeichen — der IC-Parameter

Mit dem Session-Parameter IC (Insertion Characters) geben Sie an, welche Zeichen unmittelbar *vor einem Feldwert* eingefügt werden, der von einem DISPLAY-Statement ausgegeben wird. Sie können 1 bis 10 Zeichen angeben.

Bei einem numerischen Feld werden die Einfügungszeichen unmittelbar vor die erste signifikante Stelle, die ausgegeben wird, gestellt, und zwar ohne Leerstellen zwischen dem Einfügungszeichen und dem Feldwert. Bei alphanumerischen Feldern hat der IC-Parameter die gleiche Wirkung wie der LC-Parameter.

Die Parameter LC und IC dürfen nicht beide gleichzeitig für ein Feld angegeben werden.

Der IC-Parameter kann mit den folgenden Statements verwendet werden:

- FORMAT
- DISPLAY

Der IC-Parameter kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Einfügungszeichen im Unicode-Format — der ICU Parameter

Der Session-Parameter `ICU` ist identisch mit dem Session-Parameter `IC`. Der Unterschied ist, dass alle vorangestellten Zeichen immer im Unicode-Format gespeichert werden.

Das gibt Ihnen die Möglichkeit, Einfügungszeichen mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Weitere Informationen siehe *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, Abschnitt *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

Die Parameter `LCU` und `ICU` dürfen nicht beide gleichzeitig für ein Feld angegeben werden.

Nachgestellte Zeichen — der TC-Parameter

Mit dem Session-Parameter `TC` (Trailing Characters) geben Sie an, welche Zeichen unmittelbar *hinter einem Feld* ausgegeben werden, das von einem `DISPLAY`-Statement ausgegeben wird. Die Breite der Ausgabespalte wird entsprechend vergrößert. Sie können 1 bis 10 Zeichen angeben.

Der `TC`-Parameter kann mit den folgenden Statements verwendet werden:

- `FORMAT`
- `DISPLAY`

Der `TC`-Parameter kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Vorangestellte Zeichen im Unicode-Format — der TCU Parameter

Der Session-Parameter `TCU` ist identisch mit dem Session-Parameter `TC`. Der Unterschied ist, dass alle vorangestellten Zeichen immer im Unicode-Format gespeichert werden.

Das gibt Ihnen die Möglichkeit, Einfügungszeichen mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Weitere Informationen siehe *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, Abschnitt *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

Ausgabelänge — der AL- und der NL-Parameter

Mit dem Session-Parameter `AL` bestimmen Sie die *Ausgabelänge eines alphanumerischen Feldes*; mit dem `NL`-Parameter bestimmen Sie die *Ausgabelänge eines numerischen Feldes*. Diese Länge ist die Länge, in der das Feld ausgegeben wird, und kann kürzer oder länger sein als die tatsächliche Länge des Feldes (die für ein Datenbankfeld im DDM bzw. für eine Benutzervariable im `DEFINE DATA`-Statement definiert ist).

Beide Parameter können mit den folgenden Statements verwendet werden:

- `FORMAT`
- `DISPLAY`
- `WRITE`
- `PRINT`
- `INPUT`

Sie können dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Anmerkung: Wenn eine Editiermaske angegeben ist, hat diese Vorrang vor einer `AL`- bzw. `NL`-Angabe. Editiermasken werden im Abschnitt ***Editiermasken - der EM-Parameter*** beschrieben.

Ausgabelänge — der DL Parameter

Mit dem Session-Parameter `DL` bestimmen Sie die *Ausgabelänge eines Felds des Formats A oder U*, weil die Ausgabelänge einer Unicode-Zeichenkette doppelt so lang sein kann wie die Zeichenkette und der Benutzer die Möglichkeit haben muss, die ganze Zeichenkette anzuzeigen. Der Standardwert ist zum Beispiel die Länge für ein Format/Länge U10, die Anzeigelänge kann 10 bis 20 sein, während die Standardlänge (ohne DL-Angabe) 10 ist.

Der `DL`-Parameter kann mit den folgenden Statements verwendet werden:

- `FORMAT`
- `DISPLAY`
- `WRITE`
- `PRINT`
- `INPUT`

Der `DL`-Parameter kann sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Der Unterschied zwischen den Session-Parametern `AL` und `DL` besteht darin, dass der `AL`-Parameter die Datenlänge eines Feldes bestimmt, während der `DL`-Parameter die Anzahl der Spalten bestimmt, die zur Anzeige des Feldes auf dem Schirm benutzt werden. Der Benutzer kann in Einabefeldern „blättern“, um den ganzen Inhalt eines Feldes zu sehen, wenn der mit dem `DL`-Parameter angegebene Wert kleiner als die Länge der Felddaten ist.

Anmerkung: Der `DL`-Parameter ist auch bei Feldern des Formats A zulässig. Damit kann die Edit-Control-Size kleiner als der Inhalt eines Feldes gemacht werden.

Beispiel:

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2
END
```

Das obige Programm liefert die folgende Ausgabe, in der der Inhalt des Feldes `#U2` unvollständig ist:

```
#U1 latintxt00 #U2 特別是伺服
```

Wird bei dem Feld `#U2` der Session-Parameter `DL` verwendet, dann wird der Inhalt dieses Feldes korrekt angezeigt.

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2 (DL=20)
END
```

Ergebnis:

```
#U1 latintxt00 #U2 特别是伺服器都需要支
```

Vorzeichen-Stelle — der SG-Parameter

Mit dem Session-Parameter `SG` (Sign Position) bestimmen Sie, ob numerische Felder eine zusätzliche Stelle zur Anzeige des Vorzeichens erhalten sollen.

- Standardmäßig gilt `SG=ON`, d.h. numerische Felder erhalten eine Vorzeichen-Stelle.
- Wenn Sie `SG=OFF` angeben, werden negative Werte in numerischen Feldern ohne Minuszeichen (-) ausgegeben.

Der `SG`-Parameter kann mit den folgenden Statements verwendet werden:

- `FORMAT`
- `DISPLAY`
- `PRINT`
- `WRITE`
- `INPUT`

Er kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Anmerkung: Wenn eine Editiermaske angegeben ist, hat diese Vorrang vor einer `SG`-Angabe. **Editiermasken** sind im Abschnitt *Editiermasken - der EM-Parameter* beschrieben.

Beispielprogramm ohne Parameter

```
** Example 'FORMAX03': FORMAT (without FORMAT and compare with FORMAX04)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME
          FIRST-NAME
          SALARY (1:1)
          BONUS  (1:1,1:1)
```

```
END-READ
END
```

Ausgabe des Programms `FORMAX03`:

Das obige Programm enthält keine Parameterangaben und erzeugt folgende Ausgabe:

```
Page      1                                                  04-11-11  11:11:11

        NAME                 FIRST-NAME          ANNUAL      BONUS
                                                 SALARY
-------------------- -------------------- ---------- ----------

JONES                VIRGINIA                 46000       9000
JONES                MARSHA                   50000          0
JONES                ROBERT                   31000          0
JONES                LILLY                    24000          0
JONES                EDWARD                   37600          0
```

Beispielprogramm mit Parametern AL, NL, LC, IC und TC

In diesem Beispiel werden die Session-Parameter `AL`, `NL`, `LC`, `IC` und `TC` benutzt.

```
** Example 'FORMAX04': FORMAT (with parameters AL, NL, LC, TC, IC and
**                     compare with FORMAX03)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
FORMAT AL=10 NL=6
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME            (LC=*)
          FIRST-NAME      (TC=*)
          SALARY (1:1)    (IC=$)
          BONUS (1:1,1:1) (LC=>)
END-READ
END
```

Das obige Programm erzeugt folgende Ausgabe. Vergleichen Sie sie mit der Ausgabe des vorigen Programms, um zu sehen, wie sich die einzelnen Parameter auswirken:

```
Page      1                                                04-11-11  11:11:11

   NAME     FIRST-NAME   ANNUAL   BONUS
                         SALARY
----------- ----------- -------- --------

*JONES      VIRGINIA  *   $46000 >   9000
*JONES      MARSHA    *   $50000 >      0
*JONES      ROBERT    *   $31000 >      0
*JONES      LILLY     *   $24000 >      0
*JONES      EDWARD    *   $37600 >      0
```

Wie Sie im obigen Beispiel sehen, schließt eine mit einem `AL`- oder `NL`-Parameter angegebene Ausgabelänge die mit einem `LC`-, `IC`- und `TC`-Parameter angegebenen Zeichen nicht mit ein: die Breite der `NAME`-Spalte ist z.B. 11 Stellen: 10 für den Feldwert (`AL=10`) plus 1 vorangestelltes Zeichen.

Die Spalten `SALARY` und `BONUS` sind jeweils 8 Stellen breit: 6 Stellen für den Feldwert (`NL=6`), plus 1 vorangestelltes bzw. eingefügtes Zeichen, plus 1 Vorzeichen-Stelle (da `SG=ON` gilt).

Ausgabe identischer Werte unterdrücken — der IS-Parameter

Mit dem Session-Parameter `IS` (Identical Suppress) können Sie die mehrmalige Ausgabe identischer Werte in aufeinanderfolgenden Zeilen, die von einem `WRITE`- oder `DISPLAY`-Statement erzeugt werden, unterdrücken.

- Standardmäßig gilt `IS=OFF`. Dies bedeutet, dass identische Werte angezeigt werden.
- Ist `IS=ON` gesetzt, wird ein Wert, der identisch mit dem vorherigen Wert des Feldes ist, nicht angezeigt.

Der`IS`-Parameter kann angegeben werden:

- in einem `FORMAT`-Statement; er gilt dann für den gesamten Report.
- in einem `DISPLAY`- oder `WRITE`-Statement, und zwar sowohl auf Statement- als auch auf Elementebene.

Die Wirkung des Parameters `IS=ON` kann für einen Datensatz mit dem Statement `SUSPEND IDENTICAL SUPPRESS` ausgesetzt werden. Näheres zu diesem Statement finden Sie in der *Statements*-Dokumentation.

Vergleichen Sie die Ausgaben der beiden folgenden Beispielprogramme miteinander, um die Wirkung des `IS`-Parameters zu sehen. Im zweiten Programm wird die Anzeige identischer Werte im Feld `NAME` unterdrückt.

Beispielprogramm ohne IS-Parameter

```
** Example 'FORMAX05': FORMAT (without parameter IS
**                     and compare with FORMAX06)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

Ausgabe des Programms `FORMAX05`:

```
 Page      1                                                04-11-11  11:11:11

        NAME              FIRST-NAME
 -------------------- --------------------

 JONES                VIRGINIA
 JONES                MARSHA
 JONES                ROBERT
```

Beispielprogramm mit IS-Parameter

```
** Example 'FORMAX06': FORMAT (with parameter IS
**                     and compare with FORMAX05)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FORMAT IS=ON
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

Ausgabe des Programms `FORMAX06`:

```
Page      1                                                  04-11-11  11:54:01

        NAME                FIRST-NAME
-------------------- --------------------

JONES                VIRGINIA
                     MARSHA
                     ROBERT
```

Nullwerte anzeigen — der ZP-Parameter

Mit dem Profil- und Session-Parameter `ZP` (Zero Printing) bestimmen Sie, wie ein Feldwert, der Null ist, ausgegeben wird.

- Standardmäßig gilt `ZP=ON`, d.h. für einen Feldwert, der Null ist, wird eine `0` (bei numerischen Feldern) bzw. der ganze Feldwert (bei Zeitfeldern) ausgegeben.
- Wenn Sie `ZP=OFF` angeben, wird ein Feldwert, der Null ist, gar nicht ausgegeben.

Der `ZP`-Parameter kann angegeben werden:

- in einem `FORMAT`-Statement; er gilt dann für den gesamten Report.
- in einem `DISPLAY`- oder `WRITE`-Statement, und zwar sowohl auf Statement- als auch auf Elementebene.

Vergleichen Sie die Ausgaben der beiden folgenden **Beispielprogramme** miteinander, um die Wirkung der Parameter `ZP` und `ES` zu sehen.

Leerzeilen unterdrücken — der ES-Parameter

Mit dem Parameter `ES` (Empty Line Suppression) können Sie die Ausgabe von mit einem `DISPLAY`- oder `WRITE`-Statement erzeugten Leerzeilen unterdrücken.

- Standardmäßig gilt `ES=OFF`. Dies bedeutet, dass alle Zeilen, die Leerwerte enthalten, angezeigt werden.
- Wenn Sie `ES=ON` angeben, wird eine mit einem `DISPLAY`- oder `WRITE`-Statement erzeugte Zeile, die nur Leerwerte enthält, unterdrückt. Die Verwendung des `ES`-Parameters empfiehlt sich, wenn bei der Ausgabe von multiplen Feldern oder Periodengruppen die Ausgabe vieler Leerzeilen zu erwarten ist.

Der `ES`-Parameter kann angegeben werden:

- in einem `FORMAT`-Statement; er gilt dann für den gesamten Report.

- in einem DISPLAY- oder WRITE-Statement, und zwar auf Statement-Ebene.

Anmerkung: Um die Leerwertunterdrückung auch für numerische Werte zu erhalten, muss für die betreffenden Felder neben ES=ON auch der Parameter ZP=OFF gesetzt werden, was bewirkt, dass Nullwerte in Leerwerte umgesetzt und dann ebenfalls nicht ausgegeben werden.

Vergleichen Sie die Ausgaben der beiden folgenden Beispielprogramme miteinander, um die Wirkung der Parameter ZP und ES zu sehen.

Beispielprogramm ohne Parameter ZP und ES

```
** Example 'FORMAX07': FORMAT (without parameter ES and ZP
**                      and compare with FORMAX08)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
*
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)
END-READ
END
```

Ausgabe des Programms FORMAX07:

```
Page      1                                                04-11-11  11:58:23

        NAME              FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                   9000
                                                6750
JONES                MARSHA                        0
                                                   0
JONES                ROBERT                        0
                                                   0
JONES                LILLY                         0
                                                   0
```

Beispielprogramm mit den Parametern ZP und ES

```
** Example 'FORMAX08': FORMAT (with parameters ES and ZP
**                     and compare with FORMAX07)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
*
FORMAT ES=ON
*
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)(ZP=OFF)
END-READ
END
```

Ausgabe des Programms `FORMAX08`:

```
Page      1                                              04-11-11  11:59:09

        NAME                FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                   9000
                                                6750
JONES                MARSHA
JONES                ROBERT
JONES                LILLY
```

Weitere Beispiele für Feldausgabe-relevante Parameter

Weitere Beispiele für die Parameter `LC`, `IC`, `TC`, `AL`, `NL`, `IS`, `ZP` und `ES` und das `SUSPEND IDENTICAL SUPPRESS`-Statement finden Sie in den folgenden Beispielprogrammen.

- *DISPLX17 - DISPLAY-Statement (mit NL, AL, IC, LC, TC)*
- *DISPLX18 – DISPLAY-Statement (Benutzung von Voreinstellungen für SF, AL, UC, LC, IC, TC und zum Vergleich mit DISPLX19)*
- *DISPLX19 - DISPLAY-Statement (mit SF, AL, LC, IC, TC und zum Vergleich mit DISPLX18)*
- *SUSPEX01 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP bei DISPLAY)*

- *SUSPEX02 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP in DISPLAY). Identisch mit SUSPEX01, aber mit IS=OFF.*
- *COMPRX03 - COMPRES-Statement (in Verbindung mit LC und TC)*

40 Codepage-Editiermasken — der EM-Parameter

- Verwendung des EM-Parameters 362
- Editiermasken für numerische Felder 363
- Editiermasken für alphanumerische Felder 363
- Länge der Felder 363
- Editiermasken für Datums- und Zeitfelder 364
- Trennzeichen-Angaben an lokale Standards anpassen 364
- Beispiele für Editiermasken 366
- Weitere Beispiele für Editiermasken 369

Dieses Kapitel beschreibt, wie Sie eine Editiermaske für ein alphanumerisches oder numerisches Feld angeben können.

Verwendung des EM-Parameters

Der Session-Parameter EM wird dazu verwendet, für ein numerisches oder alphanumerisches Feld eine sogenannte Editiermaske anzugeben, d.h. das Format, in dem die Feldwerte ausgegeben werden sollen, Zeichen für Zeichen festzulegen. Wenn Sie den Session-Parameter EMU verwenden, können Sie Unicode-Zeichen auf die gleiche Weise benutzen wie für den Session-Parameter EM beschrieben.

Beispiel:

```
DISPLAY NAME (EM=X^X^X^X^X^X^X^X^X^X)
```

In diesem Beispiel steht jedes X für ein Zeichen eines ausgegebenen alphanumerischen Feldwertes und jedes Circumflex (^) für eine Leerstelle. Bei Anzeige mittels DISPLAY-Statement, würde der Name JOHNSON in diesem Fall wie folgt ausgegeben:

```

J O H N S O N
```

Sie können den Session-Parameter EM an folgenden Stellen angeben:

- auf Report-Ebene (in einem FORMAT-Statement),
- auf Statement-Ebene (in einem DISPLAY-, WRITE-, INPUT-, MOVE EDITED- oder PRINT-Statement)
- oder auf Elementebene, d.h. Feldebene, (in einem DISPLAY-, WRITE- oder -Statement).

Eine mit dem EM-Parameter definierte Editiermaske hat Vorrang vor einer im **DDM** für das betreffende Feld definierten Standard-Editiermaske. Siehe auch *Using the DDM Editor Windows, Specifying Extended Field Attributes*.

Falls EM=OFF gesetzt worden ist, wird überhaupt keine Editiermaske verwendet.

Eine auf Statement-Ebene definierte Editiermaske hat Vorrang vor einer auf Programm-Ebene definierten Editiermaske.

Eine auf Feldebene definierte Editiermaske hat Vorrang vor einer auf Statement-Ebene definierten Editiermaske.

Editiermasken für numerische Felder

Bei Editiermasken für numerische Felder (Formate N, I, P, F) geben Sie für jede auszugebende Ziffer eine `9` an, und ein `Z` für jede Ziffer, die nur ausgegeben werden soll, wenn sie nicht `0` ist.

- Ein `Z` wird benutzt, um anzuzeigen, dass die Ausgabe-Position nur ausgefüllt wird, wenn die verfügbare Zahl nicht Null ist.
- Ein Dezimalkomma wird durch einen Punkt (.) angegeben.

Stellen nach dem Komma dürfen nicht mit `Z` angegeben werden. Weitere Zeichen dürfen vor oder nachgestellt oder eingefügt werden, z.B. Vorzeichen.

Weitere Informationen siehe Session-Parameter `EM`, *Editiermasken für numerische Felder* in der *Parameter-Referenz*-Dokumentation.

Editiermasken für alphanumerische Felder

Editiermasken für alphanumerische Felder müssen für jedes auszugebende alphanumerische Zeichen ein `X` enthalten.

Auch hier dürfen weitere Zeichen (bis auf einige Ausnahmen) vor,- nachgestellt oder hinzugefügt werden (in Apostrophen (') oder ohne).

Leerstellen in numerischen wie alphanumerischen Feldern werden mit einem Circumflex (^) gekennzeichnet.

Weitere Informationen siehe Session-Parameter `EM`, *Editiermasken für alphanumerische Felder* in der *Parameter-Referenz*-Dokumentation.

Länge der Felder

Wenn Sie für ein Feld eine Editiermaske definieren, beachten Sie bitte die Länge des Feldes.

- Ist die Editiermaske länger als das Feld, hat dies unvorhersehbare Auswirkungen.
- Ist die Editiermaske kürzer als das Feld, kann es sein, dass ein Feldwert nur unvollständig ausgegeben wird.

Beispiele:

Nehmen wir an, ein alphanumerisches Feld ist 12 Stellen lang und der ausgegebene Feldwert ist JOHNSON, dann würden folgende Editiermasken in folgenden Ausgaben resultieren:

Editiermaske	Ausgabe
EM=X.X.X.X.X	J.O.H.N.S
EM=****XXXXXX****	****JOHNSO**

Editiermasken für Datums- und Zeitfelder

Editiermasken für *Datumsfelder* können die Zeichen D für Tag, M für Monat und Y für Jahr in verschiedenen Kombinationen enthalten.

Editiermasken für *Zeitfelder* können die Zeichen H für Stunde, I für Minute, S für Sekunde und T für Zehntelsekunde in verschiedenen Kombinationen enthalten.

Im Zusammenhang mit Editiermasken für Datums- und Zeitfelder siehe auch die Datums- und Uhrzeit-Systemvariablen.

Trennzeichen-Angaben an lokale Standards anpassen

Natural-Programme werden in der ganzen Welt in Geschäftsanwendungen eingesetzt. Je nach den lokalen Gegebenheiten ist es üblich, numerische Datenfelder und Felder mit einer Datums- oder Zeitangabe bei der Anzeige in Eingabe/Ausgabe-Statements in einem ganz bestimmten Format auszugeben. Die unterschiedliche Erscheinungsform sollte nicht durch einen anderen Programmcode realisiert werden, der selektiv als eine Funktion des Bereichs verarbeitet wird, in dem das Programm ausgeführt wird, sondern sollte mit demselben Programmtyp in Verbindung mit einer Reihe von Laufzeit-Parametern ausgeführt werden, um das Dezimalpunkt-Zeichen und das „Tausender-Trennzeichen" anzugeben.

Folgende Themen werden behandelt:

- Dezimaltrennzeichen
- Dynamisches Tausendertrennzeichen

- Beispiele

Dezimaltrennzeichen

Der Natural-Parameter DC (Dezimaltrennzeichen) steht zur Verfügung, um das Zeichen anzugeben, das anstelle von Zeichen eingefügt wird, die zur Darstellung des Dezimal-Trennzeichens (auch als „Basiszeichen" bezeichnet) in Editiermasken benutzt werden. Dieser Parameter ermöglicht es den Benutzern eines Natural-Programms oder einer Natural-Anwendung, beliebige Zeichen oder Sonderzeichen zu wählen, um die Ganzzahl-Stellen von den Dezimalstellen eines numerischen Datenelements zu trennen, und ermöglicht es zum Beispiel US-Unternehmen, den Dezimalpunkt (.) zu verwenden, und europäischen Unternehmen, das Komma (,) zu benutzen.

Dynamisches Tausendertrennzeichen

Um die Ausgabe von großen Ganzzahl-Werten zu strukturieren, ist es üblich, Trennzeichen zwischen jeder dritten Ziffer einer Ganzzahl einzufügen, um Tausender voneinander zu trennen. Dieses Trennzeichen wird als Tausendertrennzeichen bezeichnet. Beispielsweise kann in den Vereinigten Staaten und Großbritannien ein Komma für diesem Zweck benutzt (1,000,000.00) werden, wohingegen in Deutschland und Österreich das Leerzeichen (1'000'000,00) oder der Punkt (1.000.000,00) und in der Schweiz und Liechtenstein das Hochkomma (1'000'000,00) verwendet werden kann.

In einer Natural-Editiermaske ist ein dynamisches Tausendertrennzeichen ein Komma (oder Punkt), welches die Position anzeigt, an der (mit dem Parameter THSEPCH definierte) Tausendertrennzeichen zur Laufzeit eingefügt werden. Zur Kompilierungszeit aktiviert oder deaktiviert der Natural-Profilparameter THSEP oder die Option THSEP des Systemkommandos COMPOPT die Interpretation des Kommas (oder Punktes) als ein dynamisches Tausendertrennzeichen.

Wenn THSEP auf OFF (Voreinstellung) gesetzt ist, wird jedes in der Editiermaske als Tausendertrennzeichen benutzte Zeichen als Literal behandelt und zur Laufzeit unverändert angezeigt. Diese Einstellung gewährleistet die Abwärtskompatibilität.

Wenn THSEP auf ON gesetzt ist, wird ein Komma (oder Punkt) in der Editiermaske als dynamisches Tausendertrennzeichen interpretiert. Im Allgemeinen ist das dynamische Tausendertrennzeichen ein Komma, aber wenn das Komma bereits als Dezimalzeichen (DC) vergeben ist, wird der Punkt als dynamisches Trennzeichen verwendet.

Zur Laufzeit werden die dynamischen Tausendertrennzeichen durch den aktuellen Wert des Parameters THSEPCH (Tausendertrennzeichen) ersetzt.

Beispiele

Ein Natural-Programm, das mit den Parameter-Einstellungen `DC='.'` und `THSEP=ON` katalogisiert ist, benutzt die Editiermaske `(EM=ZZ,ZZZ,ZZ9.99)`.

Parameter-Einstellungen zur Laufzeit	Wird angezeigt als
`DC='.'` und `THSEPCH=','`	`1,234,567.89`
`DC=','` und `THSEPCH='.'`	`1.234.567,89`
`DC=','` und `THSEPCH='/'`	`1/234/567,89`
`DC=','` und `THSEPCH=' '`	`1 234 567,89`
`DC=','` und `THSEPCH=''''`	`1'234'567,89`

Beispiele für Editiermasken

Im folgenden sehen Sie einige Beispiele für Editiermasken und die Ausgaben, die sie erzeugen.

Zusätzlich ist die jeweilige Kurzschreibweise angegeben. Sie können die Kurz- oder Langschreibweise wahlweise verwenden.

Editiermaske	Kurzschreibweise	Ausgabe A	Ausgabe B
EM=999.99	EM=9(3).9(2)	367.32	005.40
EM=ZZZZZ9	EM=Z(5)9(1)	0	579
EM=X^XXXXX	EM=X(1)^X(5)	B LUE	A 19379
EM=XXX...XX	EM=X(3)...X(2)	BLU...E	AAB...01
EM=MM.DD.YY	*	01.05.87	12.22.86
EM=HH.II.SS.T	**	08.54.12.7	14.32.54.3

[*] Verwenden Sie eine Datums-Systemvariable.

[**] Verwenden Sie eine Uhrzeit-Systemvariable.

Weitere Informationen zu Editiermasken finden Sie unter Session-Parameter `EM` in der *Parameter-Referenz*-Dokumentation.

Beispielprogramm ohne EM-Parameter

```
** Example 'EDITMX01': Edit mask (using default edit masks)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:3)
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E'    NAME         /
          'OCCUPATION' JOB-TITLE
          'SALARY'     SALARY (1:3)
          'LOCATION'   CITY
  SKIP 1
END-READ
END
```

Ausgabe des Programms `EDITMX01`:

Es erzeugt die folgende Ausgabe unter Verwendung von Standard-Editiermasken (soweit vorhanden):

```
Page      1                                            04-11-11  14:15:54

         N A M E            SALARY         LOCATION
       OCCUPATION
------------------------- ---------- --------------------

JONES                          46000 TULSA
MANAGER                        42300
                               39300

JONES                          50000 MOBILE
DIRECTOR                       46000
                               42700

JONES                          31000 MILWAUKEE
PROGRAMMER                     29400
                               27600
```

Beispielprogramm mit EM-Parametern

```
** Example 'EDITMX02': Edit mask (using EM)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
 DISPLAY 'N A M E'    NAME         (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X) /
                       FIRST-NAME   (EM=...X(10)...)
          'OCCUPATION' JOB-TITLE    (EM=' ___ 'X(12))
          'SALARY'     SALARY (1:3) (EM=' USD 'ZZZ,999)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `EDITMX02`:

Vergleichen Sie sie mit der des vorigen Programms (*Beispielprogramm ohne EM-Parameter*), um zu sehen, wie sich die `EM`-Angaben auf die Anzeige der Felder auswirken:

```
Page      1                                                04-11-11  14:15:54

          N A M E                OCCUPATION      SALARY
         FIRST-NAME
----------------------------- ---------------- -----------

J O N E S                     ___ MANAGER      USD  46,000
..VIRGINIA  ...                                USD  42,300
                                               USD  39,300

J O N E S                     ___ DIRECTOR     USD  50,000
..MARSHA    ...                                USD  46,000
                                               USD  42,700

J O N E S                     ___ PROGRAMMER   USD  31,000
..ROBERT    ...                                USD  29,400
                                               USD  27,600
```

Weitere Beispiele für Editiermasken

Siehe die folgenden Beispiel-Programme.

- *EDITMX03 - Editiermaske (unterschiedliche EM-Angabe bei alphanumerischen Feldern)*
- *EDITMX04 - Editiermaske (unterschiedliche EM-Angaben bei numerischen Feldern)*
- *EDITMX05 - Editiermaske (EM-Angaben für Datums- und Uhrzeit-Systemvariablen)*

41 Unicode-Editiermasken — EMU-Parameter

Unicode-Editiermasken können auf die gleiche Weise wie Codepage-Editiermasken verwendet werden. Der Unterschied besteht darin, dass die Editiermaske immer im Unicode-Format gespeichert wird.

Das gibt Ihnen die Möglichkeit, Editiermasken mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Für die Benutzung der Unicode-Editiermasken gilt dasselbe wie im Abschnitt ***Edit Masks - EM Parameter*** beschrieben.

Informationen zum Session-Parameter `EMU`, finden Sie im Abschnitt *EMU - Unicode Edit Mask* (in der *Parameter-Referenz*).

42 Vertikale Ausgabe von Feldwerten

- Vertikale Ausgaben erzeugen 374
- Vertikale Ausgabe durch Kombination von DISPLAY und WRITE 374
- Tabulator-Notation — T*field 375
- Positionierungsnotation x/y 376
- DISPLAY VERT-Statement 377
- Weiteres Beispiel für DISPLAY VERT- mit WRITE-Statement 383

Dieses Kapitel beschreibt, wie Sie die Funktionen der Statements DISPLAY und WRITE miteinander kombinieren können, um vertikale Ausgaben von Feldwerten zu erzeugen.

Vertikale Ausgaben erzeugen

Natural bietet Ihnen zwei Möglichkeiten, die verschiedenen Daten eines Datensatzes bei der Ausgabe untereinander anzuordnen:

- mit einer Kombination von DISPLAY- und WRITE-Statement,
- mit der VERT-Klausel im DISPLAY-Statement.

Vertikale Ausgabe durch Kombination von DISPLAY und WRITE

Wie weiter oben beschrieben, erzeugt das DISPLAY-Statement normalerweise Ausgaben in Spaltenform mit Standardüberschriften, während das WRITE-Statement die Daten nebeneinander ohne Überschriften anordnet.

Sie können die Merkmale dieser beiden Statements miteinander verbinden, um eine vertikale Ausgabe von Feldwerten zu erzeugen.

Das DISPLAY-Statement ordnet die Werte eines Feldes untereinander an, und zwar Datensatz für Datensatz; die verschiedenen Felder eines Datensatzes werden nebeneinander ausgegeben.

Durch ein dem DISPLAY-Statement nachgestelltes WRITE-Statement haben Sie die Möglichkeit, in einem WRITE-Statement angegebene Text und/oder Feldwerte zwischen den einzelnen mit dem DISPLAY-Statement ausgegebenen Datensätzen einzufügen.

Das folgende Programm zeigt diese Kombination von DISPLAY und WRITE:

```
** Example 'WRITEX04': WRITE (in combination with DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CITY
  2 DEPT
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
  WRITE 22T 'DEPT:' DEPT
```

```
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX04`:

```
Page      1                                            04-11-11  14:15:55

        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

Tabulator-Notation — T*field

Im vorigen Beispiel ergibt sich die Position des Feldes `DEPT` aus der Tabulator-Notation *n*`T` (in diesem Fall `20T`, d.h. die Ausgabe beginnt in der 20. Bildschirmspalte).

Durch die Notation `T*`*`field`* können Sie die `WRITE`-Ausgabe nach der Position eines im vorangegangenen `DISPLAY`-Statement ausgegebenen Feldes ausrichten (wobei *`field`* der Name des Feldes ist, nach dem die Ausrichtung erfolgen soll).

Im folgenden Beispiel wird die Position der vom `WRITE`-Statement erzeugten Ausgabe mittels der Notation `T*JOB-TITLE` nach der Position des Feldes `JOB-TITLE` ausgerichtet:

```
** Example 'WRITEX05': WRITE  (in combination with DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
```

```
  WRITE T*JOB-TITLE 'DEPT:' DEPT
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX05`:

```
Page      1                                                      04-11-11  14:15:55

        NAME                    CURRENT
                               POSITION
-------------------- -------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

Positionierungsnotation x/y

Wenn bei der Kombination von `DISPLAY` und `WRITE` die vom `WRITE`-Statement erzeugte Ausgabe über mehrere Zeilen und/oder Spalten gehen soll, empfiehlt sich die Verwendung der Notation *x/y* (Zahl-Schrägstrich-Zahl), mit der Sie angeben können, in welcher Zeile/Spalte etwas ausgegeben werden soll. Die Zahl vor dem Schrägstrich gibt die Zeile an, die Zahl hinter dem Schrägstrich die Spalte.

Das folgende Programm veranschaulicht die Verwendung dieser Notation:

```
** Example 'WRITEX06': WRITE  (with n/n)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
  2 ADDRESS-LINE (1:1)
  2 CITY
  2 ZIP
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
```

```
  DISPLAY 'NAME AND ADDRESS' NAME
  WRITE   1/5  FIRST-NAME
          1/30 MIDDLE-I
          2/5  ADDRESS-LINE (1:1)
          3/5  CITY
          3/30 ZIP /
END-READ
END
```

Ausgabe des Programms WRITEX06:

```
Page      1                                                   04-11-11  14:15:55

  NAME AND ADDRESS
--------------------

RUBIN
    SYLVIA                   L
    2003 SARAZEN PLACE
    NEW YORK                 10036

WALLACE
    MARY                     P
    12248 LAUREL GLADE C
    NEW YORK                 10036

KELLOGG
    HENRIETTA                S
    1001 JEFF RYAN DR.
    NEWARK                   19711
```

DISPLAY VERT-Statement

Standardmäßig gibt Natural die Felder nebeneinander aus.

Mit der VERT-Klausel können Sie erreichen, dass bei einem DISPLAY-Statement die Werte der verschiedenen Felder eines Datensatzes nicht nebeneinander, sondern untereinander (in vertikaler Anordnung) ausgegeben werden.

Mit einer HORIZ-Klausel können Sie dies im selben DISPLAY-Statement wieder rückgängig machen und zur horizontalen Ausgabe zurückkehren.

Die Ausgabe von Spaltenüberschriften beim DISPLAY VERT wird über eine AS-Klausel gesteuert:

- Ohne AS-Klausel werden keine Spaltenüberschriften ausgegeben. Siehe **Beispiel 1**.
- AS CAPTIONED bewirkt die Ausgabe der Standard-Spaltenüberschriften. Siehe **Beispiel 2**.

- `AS 'text'` bewirkt, dass Text als Spaltenüberschrift ausgegeben wird. Beachten Sie hierbei, dass ein Schrägstrich (/) innerhalb von Text-Elementen eines `DISPLAY`-Statements einen Zeilenvorschub bewirkt. Siehe **Beispiel 3**.
- `AS 'text' CAPTIONED` bewirkt, dass Text als Spaltenüberschrift ausgegeben wird und außerdem die Standard-Spaltenüberschrift in jeder Ausgabezeile dem jeweiligen Feldwert direkt vorangestellt wird. Siehe **Beispiel 4**.

Beispiel 1 - DISPLAY VERT ohne AS-Klausel

Das folgende Programm verwendet keine `AS`-Klausel, d.h. es werden keine Spaltenüberschriften ausgegeben.

```
** Example 'DISPLX09': DISPLAY  (without column title)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT NAME FIRST-NAME / CITY
  SKIP 2
END-READ
END
```

Ausgabe des Programms `DISPLX09`:

Beachten Sie, dass alle Feldwerte vertikal, d.h. untereinander, ausgegeben werden:

```
Page      1                                                   04-11-11  14:15:54


RUBIN
SYLVIA

NEW YORK


WALLACE
MARY

NEW YORK


KELLOGG
```

```
HENRIETTA

NEWARK
```

Beispiel 2 - DISPLAY VERT AS CAPTIONED und HORIZ

Das folgende Programm enthält eine `VERT`- und eine `HORIZ`-Klausel, die bewirken, dass einige Ausgaben vertikal und andere horizontal angeordnet sind, sowie eine `AS CAPTIONED`-Klausel zur Ausgabe der Standard-Spaltenüberschriften.

```
** Example 'DISPLX10': DISPLAY (with VERT as CAPTIONED and HORIZ clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS CAPTIONED NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX10`:

```
Page      1                                                 04-11-11  14:15:54

        NAME                  CURRENT            ANNUAL
     FIRST-NAME              POSITION            SALARY
-------------------- ------------------------- ----------

RUBIN                SECRETARY                      17000
SYLVIA

WALLACE              ANALYST                        38000
MARY

KELLOGG              DIRECTOR                       52000
HENRIETTA
```

Beispiel 3 - DISPLAY VERT AS 'text'

Das folgende Programm enthält eine `AS 'text'`-Klausel, die bewirkt, dass der angegebene `'text'` als Spaltenüberschrift ausgegeben wird.

Anmerkung: Ein Schrägstrich (/) in dem Textelement in einem `DISPLAY`-Statement bewirkt einen Zeilenumbruch.

```
** Example 'DISPLX11': DISPLAY  (with VERT AS 'text' clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX11`:

```
Page      1                                                04-11-11  14:15:54

     EMPLOYEES                CURRENT            ANNUAL
                             POSITION           SALARY
-------------------- ------------------------- ----------

RUBIN                SECRETARY                     17000
SYLVIA

WALLACE              ANALYST                       38000
MARY

KELLOGG              DIRECTOR                      52000
HENRIETTA
```

Beispiel 4 - DISPLAY VERT AS 'text' CAPTIONED

Die Klausel `AS 'text' CAPTIONED` bewirkt, dass der angegebene Text als Spaltenüberschrift angezeigt wird und dass die Standard-Spaltenüberschriften direkt vor dem Feldwert in jeder ausgegebenen Zeile angezeigt werden:

Das folgende Programm enthält eine `AS 'text' CAPTIONED`-Klausel.

```
** Example 'DISPLX12': DISPLAY (with VERT AS 'text' CAPTIONED clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' CAPTIONED NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX12`:

Diese Klausel bewirkt, dass die Standard-Spaltenüberschriften (`NAME` und `FIRST-NAME`) vor den Feldwerten ausgegeben werden:

```
Page      1                                            04-11-11  14:15:54

           EMPLOYEES                      CURRENT           ANNUAL
                                         POSITION           SALARY
------------------------------- ------------------------- ----------

NAME RUBIN                      SECRETARY                      17000
FIRST-NAME SYLVIA

NAME WALLACE                    ANALYST                        38000
FIRST-NAME MARY

NAME KELLOGG                    DIRECTOR                       52000
FIRST-NAME HENRIETTA
```

Tabulator-Notation P*field

Bei einer Kombination von DISPLAY VERT-Statement mit nachfolgendem WRITE-Statement können Sie mit der Notation P**field-name* die Feld-Ausgabe des WRITE-Statements nach der Zeilen und Spalten-Position eines im DISPLAY VERT-Statements angegebenen Feldes ausrichten.

Im folgenden Programm werden die Felder SALARY und BONUS in der gleichen Spalte ausgegeben, SALARY in jeder ersten Zeile, BONUS in jeder zweiten Zeile.

Der Text ***SALARY PLUS BONUS*** ist nach SALARY ausgerichtet, d.h. der Text wird in der gleichen Spalte wie SALARY und in der ersten Zeile ausgegeben. Der Text (IN US DOLLARS) hingegen ist nach BONUS ausgerichtet; entsprechend wird dieser Text in der gleichen Spalte wie BONUS und in der zweiten Zeile ausgegeben.

```
** Example 'WRITEX07': WRITE  (with P*field)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'LOS ANGELES'
  DISPLAY NAME JOB-TITLE
          VERT AS 'INCOME' SALARY (1) BONUS (1,1)
  WRITE P*SALARY '***SALARY PLUS BONUS***'
        P*BONUS  '(IN US DOLLARS)'
  SKIP 1
END-READ
END
```

Ausgabe des Programms WRITEX07:

```
Page      1                                                  04-11-11  14:15:55

        NAME                   CURRENT            INCOME
                              POSITION
-------------------- ------------------------- ----------

SMITH                                                   0
                                                        0
                                               ***SALARY PLUS BONUS***
                                               (IN US DOLLARS)

POORE JR             SECRETARY                      25000
```

```
                                                            0
                                                   ***SALARY PLUS BONUS***
                                                   (IN US DOLLARS)

PREPARATA            MANAGER                           46000
                                                        9000
                                                   ***SALARY PLUS BONUS***
                                                   (IN US DOLLARS)
```

Weiteres Beispiel für DISPLAY VERT- mit WRITE-Statement

Siehe das folgende Beispiel-Programm:

- ***WRITEX10 - WRITE-Statement (mit nT, T*field und P*field)***

43 Weitere Programmieraspekte

In diesem Teil werden folgende Themen behandelt:

- Ende eines Statements, Programms oder einer Anwendung
- Verarbeitung von Anwendungsfehlern
- Bedingte Verarbeitung – IF-Statement
- Schleifenverarbeitung
- Gruppenwechsel
- Datenberechnungen
- Systemvariablen und Systemfunktionen
- Stack
- Verarbeitung von Datumsinformationen
- Text-Notation
- Benutzer-Kommentare
- Logische Bedingungen
- Regeln für arithmetische Operationen
- Natural-Subprogramme aus 3GL-Programmen aufrufen
- Betriebssystemkommandos aus einem Natural-Programm absetzen
- Statements für den Internet- und XML-Zugriff

44 Ende eines Statements, Programms oder einer Anwendung

- Ende eines Statements 388
- Ende eines Programms 388
- Ende einer Anwendung 388

Ende eines Statements

Um das Ende eines Statements explizit zu markieren, fügen Sie ein Semikolon (;) zwischen diesem Statement und dem nächsten Statement ein. Dies dient dazu, die Programm-Struktur klarer zu gestalten, ist aber nicht erforderlich.

Ende eines Programms

Das `END`-Statement dient dazu, das Ende eines Programms, einer Function, eines Subprogramms, einer externen Subroutine bzw. einer Helproutine zu kennzeichnen.

Jedes dieser Objekte muss als letztes Statement ein `END`-Statement enthalten.

Jedes Objekt darf nur ein `END`-Statement enthalten.

Ende einer Anwendung

Ausführung einer Anwendung mit einem STOP-Statement beenden

Das `STOP`-Statement dient dazu, die Ausführung einer Natural-Anwendung abzubrechen. Ganz gleich, wo ein `STOP`-Statement in einer Anwendung ausgeführt wird, beendet es sofort die Ausführung der gesamten Anwendung.

Ausführung einer Anwendung mit einem TERMINATE-Statement beenden

Das `TERMINATE`-Statement bricht die Ausführung der Natural-Anwendung ab und beendet die Natural-Session.

Eine laufende Natural-Anwendung unterbrechen

Der Benutzer muss während der Entwicklung einer Natural-Anwendung und in Test-Situationen in der Lage sein, eine laufende Anwendung, die z.B. wegen einer Endlosschleife nicht mehr reagiert, zu unterbrechen. Um den Abbruch der Natural-Session zu vermeiden, kann die laufende Natural-Anwendung mittels der typischen Unterbrechungstastenkombination (z.B. CTRL+BREAK bei Windows, CTRL+C bei UNIX) unterbrochen werden. Daraufhin wird der Natural-Fehler NAT1016 ausgegeben und die Laufzeitfehlerverarbeitung aktiviert. Der Fehler kann mit Hilfe einer `ON ERROR`-Verarbeitung behandelt werden.

In einer Produktionsumgebung wird man diese Funktion üblicherweise ausschalten, weil es nicht mehr nötig ist, dass die Ausführung der Anwendung nach einer Unterbrechung durch einen Benutzer an einer willkürlichen Programmstelle wieder aufgenommen wird.

Mit dem Natural-Profilparameter `RTINT` wird festgelegt, ob Interrupts zulässig sind. Standardmäßig sind Interrupts nicht zulässig.

Wird dieser Parameter auf `ON` gesetzt, ist es möglich, eine laufende Natural-Anwendung mit Unterbrechungstastenkombination des Betriebssystems, z.B. CTRL+BREAK bei Windows, CTRL+C bei UNIX, zu unterbrechen. Die Tastenkombination kann mit dem Kommando `stty` geändert werden.

Natural fängt die Interrupt-Anforderung ab und bietet dem Benutzer dann in einem Fenster folgende Möglichkeiten zur Auswahl an:

- Standardmäßige Fehlerverarbeitung nach Ausgabe von NAT1016 durchführen.
- Mit der Anwendung fortfahren (Interrupt löschen)

Anmerkung: Die Natural-Anwendung kann nur unterbrochen werden, wenn die Anwendung oder Natural Studio (wenn die Anwendung dort gestartet wurde) den Input-Focus hat.

45 Verarbeitung von Anwendungsfehlern

- Naturals Standard-Fehlerverarbeitung 392
- Anwendungsspezifische Fehlerverarbeitung 393
- Verwendung eines ON ERROR-Statement-Blocks 393
- Verwendung eines Fehlertransaktionsprogramm 394
- Funktionalität für die Fehlerverarbeitung 398

Dieser Abschnitt beschreibt die zwei grundlegenden Verfahren, die Natural für die Behandlung von Anwendungsfehlern bietet: Standardverarbeitung und anwendungsspezifische Verarbeitung. Darüber hinaus beschreibt es, auf welche Weise die anwendungsspezifische Verarbeitung von Fehlern ermöglicht werden kann: durch das Kodieren eines `ON ERROR`-Statement-Blocks innerhalb eines Programmierobjekts oder durch den Einsatz eines separaten Fehlertransaktionsprogramms.

Schließlich enthält dieser Abschnitt noch eine Übersicht über die Natural-Funktionalität, mit der Sie die Fehlerverarbeitung durch Natural konfigurieren, Informationen über einen Fehler abrufen oder einen Anwendungsfehler verarbeiten oder bereinigen können.

Informationen zur Behandlung von Fehlern in einer Natural RPC-Umgebung siehe *Handling Errors* in the *Natural Remote Procedure Call*-Dokumentation.

Naturals Standard-Fehlerverarbeitung

Wenn in einer Natural-Anwendung ein Fehler auftritt, geht Natural standardmäßig folgendermaßen vor:

1. Natural beendet die Ausführung des zurzeit laufenden Anwendungsobjekts;
2. Natural gibt eine Fehlermeldung aus;
3. Natural kehrt zum Kommandoeingabe-Modus zurück.

„Kommandoeingabe-Modus“ bedeutet, dass in Abhängigkeit von Ihrer jeweiligen Natural-Konfiguration das Natural-Hauptmenü, die `NEXT`-Zeile oder ein benutzerdefiniertes Einstiegsmenü erscheinen kann.

Die angezeigte Fehlermeldung enthält die Natural-Fehlernummer, den zugehörigen Meldungstext sowie das betroffene Natural-Objekt und die Nummer der Zeile, in der der Fehler aufgetreten ist.

Da die Ausführung des Anwendungsobjekts beendet wird, kann der Status von anhängigen Datenbanktransaktionen von Maßnahmen betroffen sein, die durch die Einstellung des Profilparameters `ETEOP` bedingt sind. Falls Natural (infolge der Einstellungen dieser Parameter) kein `END TRANSACTION`-Statement ausgegeben hat, dann wird bei der Rückkehr in den Kommandoeingabe-Modus ein `BACKOUT TRANSACTION`-Statement ausgegeben.

Anwendungsspezifische Fehlerverarbeitung

Wenn die Standard-Fehlerverarbeitung nicht den Erfordernissen Ihrer Anwendung entspricht, können Sie die Verarbeitung anwendungsspezifisch anpassen. Mögliche Gründe hierfür können beispielsweise sein:

- Informationen zum Fehler sollen zwecks weiterer Untersuchung durch den Anwendungsentwickler gespeichert werden.
- Die Ausführung der Anwendung soll, falls möglich, nach der Fehlerbehebung fortgesetzt werden.
- Es ist eine spezifische Transaktionsbehandlung nötigt.

Da nach Auftreten eines Fehlers die Ausführung des betroffenen Anwendungsobjekts beendet wird, kann der Status von anhängigen Datenbanktransaktionen von Maßnahmen betroffen sein, die durch die des Profilparameters `ETEOP` ausgelöst werden. Deshalb muss die weitere Transaktionsbehandlung (mittels `END TRANSACTION`- oder `BACKOUT TRANSACTION`-Statement) über die Fehlerverarbeitung der Anwendung erfolgen.

Um eine anwendungsspezifische Fehlerverarbeitung zu ermöglichen, haben Sie folgende Möglichkeiten:

- Sie können innerhalb eines Programmierobjekts einen `ON ERROR`-Statement-Block kodieren.
- Sie können ein separates Fehlertransaktionsprogramm verwenden.

Diese Möglichkeiten werden in den folgenden Abschnitten behandelt.

Verwendung eines ON ERROR-Statement-Blocks

Sie können das `ON ERROR`-Statement verwenden, um zur Ausführungszeit auftretende Fehler in einer Anwendung an der Stelle abzufangen, an der ein Fehler auftritt.

Aus einem solchen `ON ERROR`-Statement-Block heraus kann die Ausführung der Anwendung auf der aktuellen Ebene oder auf einer übergeordneten Ebene wieder aufgenommen werden.

Außerdem können Sie ein `ON ERROR`-Statement in mehreren Objekten einer Anwendung angeben, um Fehler, die auf untergeordneten Ebenen aufgetreten sind, zu verarbeiten. Auf diese Weise können Sie die Fehlerverarbeitung exakt auf die Erfordernisse Ihrer Anwendung zuschneiden.

Verlassen eines `ON ERROR`-Statement-Blocks

Um einen `ON ERROR`-Statement-Block zu verlassen, können Sie eines der folgenden Statements angeben:

- `RETRY`

  Die Ausführung der Anwendung wird auf der aktuellen Ebene wieder aufgenommen.

- `ESCAPE ROUTINE`

  Es wird davon ausgegangen, dass die Fehlerverarbeitung abgeschlossen ist, und die Ausführung der Anwendung wird auf der übergeordneten Ebene wieder aufgenommen.

- `FETCH`

  Es wird davon ausgegangen, dass die Fehlerverarbeitung abgeschlossen ist, und das beim `FETCH`-Statement angegebene Programm wird ausgeführt.

  `STOP`

  Natural stoppt die Ausführung des betroffenen Programms, beendet die Anwendung und kehrt zum Kommandoeingabe-Modus zurück.

- `TERMINATE`

  Die Ausführung der Natural-Anwendung wird gestoppt, und die Natural-Session wird beendet.

Fehlerverarbeitungsregeln

- Wird die Ausführung des `ON ERROR`-Statement-Blocks nicht durch eines der oben genannten Statements beendet, dann wird der Fehler an das Natural-Objekt auf der übergeordneten Ebene durchgereicht, damit er durch einen dort vorhandenen `ON ERROR`-Statement-Block verarbeitet wird.
- Falls keines der Objekte auf einer der übergeordneten Ebenen einen `ON ERROR`-Statement-Block enthält, falls aber ein Fehlertransaktionsprogramm (wie im folgenden **Abschnitt** beschrieben) angegeben ist, erhält dieses Fehlertransaktionsprogramm die Kontrolle.
- Falls keines der Objekte auf einer der übergeordneten Ebenen einen `ON ERROR`-Statement-Block enthält und falls dort kein Fehlertransaktionsprogramm angegeben ist, dann greift die Standard-Fehlerverarbeitung von Natural wie **oben** beschrieben.

Verwendung eines Fehlertransaktionsprogramm

Sie können an den folgenden Stellen ein Fehlertransaktionsprogramm angeben:

- Im Profilparameter `ETA`.
- Im Natural Security Library Profile, falls Natural Security installiert ist; siehe *Components of a Library Profile* in der *Natural Security*-Dokumentation.

- Innerhalb eines Natural-Objekts, indem Sie dort mittels eines ASSIGN-, COMPUTE- oder MOVE-Statements den Namen des Fehlertransaktionsprogramms der Systemvariablen *ERROR-TA als Wert zuordnen.

Wenn Sie während der Natural-Session den Namen eines Fehlertransaktionsprogramms der Systemvariablen *ERROR-TA zuweisen, dann wird durch diese Zuweisung ein mittels Profilparameter ETA angegebenes Fehlertransaktionsprogramm ersetzt. Aber ganz gleich, ob Sie den Profilparameter ETA verwenden oder der Systemvariablen *ERROR-TA einen Wert zuweisen, die Namen von Fehlertransaktionsprogramm werden nicht gespeichert und werden von Natural nicht für die verschiedenen Ebenen der Aufruf-Hierarchie wieder hergestellt. Darum wird, wenn Sie den Namen des Programms in einem Natural-Objekt der Systemvariablen *ERROR-TA zuweisen, dieses Programm aufgerufen, um jeden Fehler zu verarbeiten, der nach dieser Zuweisung in der aktuellen Natural-Session auftritt.

Einerseits wird also, wenn Sie ein Fehlertransaktionsprogramm mit dem Profilparameter ETA angeben, eine Fehlertransaktion für die ganze Natural-Session definiert, ohne dass die Notwendigkeit besteht, innerhalb von Natural-Objekten Einzelzuweisungen vorzunehmen. Anderseits bietet das Verfahren, ein Programm der Systemvariablen *ERROR-TA zuzuweisen, mehr Flexibilität und gestattet es Ihnen zum Beispiel, in verschiedenen Zweigen der Anwendung verschiedene Fehlertransaktionsprogramme zu benutzen.

Wenn die Systemvariable *ERROR-TA zurückgesetzt (leer) wird, dann wird wie **oben** beschrieben Naturals Standard-Fehlerverarbeitung durchgeführt.

Wenn ein Fehlertransaktionsprogramm angegeben ist und ein Anwendungsfehler auftritt, wird die Ausführung der Anwendung beendet. Das angegebene Fehlertransaktionsprogramm erhält dann die Kontrolle, um eine der folgenden Maßnahmen auszuführen:

- Analyse des Fehlers
- Protokollieren der Fehlerinformationen
- Beenden der Natural-Session
- Fortsetzen der Anwendungsausführung durch Aufrufen eines Programms mittels FETCH-Statement.

Da das Fehlertransaktionsprogramm die Kontrolle so erhält, als wenn es im Kommandoeingabe-Modus eingegeben worden wäre, ist es nicht möglich, die Ausführung der Anwendung in einem der Natural-Objekte, die zum Zeitpunkt des Auftretens des Fehlers aktiv waren, wieder aufzunehmen.

Wenn der Natural-Profilparameter SYNERR auf ON gesetzt ist und zur Laufzeit ein Syntaxfehler auftritt, erhält das Fehlertransaktionsprogramm auch die Kontrolle.

Fehlertransaktionsprogramme müssen sich in einer Library befinden, in der Sie zurzeit angemeldet sind, oder in einer aktuellen Steplib Library.

Wenn ein Fehler auftritt, führt Natural ein STACK TOP DATA-Statement aus und legt folgende Informationen oben auf dem Stack ab:

Feldinhalt		Format/Länge
Natural-Fehlernummer		N4, wenn der Session-Parameter SG auf OFF gesetzt ist; N5, wenn SG=ON
Nummer der Zeile, in welcher der Fehler aufgetreten ist		N4
Status Code:		A1
C	Kommandoverarbeitungsfehler (Die Zeilennummer ist dann 0.)	
L	Logon-Verarbeitungsfehler (Die Zeilennummer ist dann 0.)	
O	Objekt-(Ausführungs-)Zeitfehler	
R	Fehler auf einem Remote Server (in Verbindung mit dem Natural RPC)	
S	Nicht korrigierbarer Syntaxfehler	
Name des Natural-Objekts, in dem der Fehler aufgetreten ist		A8
Nummer der Programmebene des Natural-Objekts, auf der der Fehler aufgetreten ist; wenn die Nummer größer als 99 ist, wird der Wert 99 auf den Stack gelegt.		N2

Wenn der Natural-Profilparameter SYNERR auf OFF gesetzt ist, wird folgende Information zusätzlich auf den Stack gelegt:

Feldinhalt	Format/Länge
Nummer der Programmebene des Natural-Objekts, auf der der Fehler aufgetreten ist; wenn die Nummer kleiner als oder gleich 99 ist, dann ist der Feldinhalt identisch mit dem des **zuvor** beschriebenen Felds mit Format/Länge N2.	I4

Wenn der Natural-Profilparameter SYNERR auf ON gesetzt ist und zur Laufzeit ein Syntaxfehler auftritt, dann ist die Nummer der Programmebene Null und die folgenden Informationen werden zusätzlich auf den Stack gelegt:

Feldinhalt	Format/Länge
Position des den Fehler verursachenden Bestandteils in der Source-Zeile	N3
Länge des verursachenden Bestandteils	N3

Diese Informationen können in einem Fehlertransaktionsprogramm mittels eines INPUT-Statements abgefragt werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #ERROR-NR           (N5)
1 #LINE               (N4)
1 #STATUS-CODE        (A1)
1 #PROGRAM            (A8)
1 #LEVEL              (N2)
1 #LEVELI4            (I4)
1 #POSITION-IN-LINE   (N3)
1 #LENGTH-OF-ERRTOKEN (N3)
END-DEFINE
IF *DATA > 6 THEN       /* SYNERR = ON and a syntax error occurred
  INPUT
    #ERROR-NR
    #LINE
    #STATUS-CODE
    #PROGRAM
    #LEVEL
    #POSITION-IN-LINE
    #LENGTH-OF-ERRTOKEN
ELSE
  INPUT                 /* other error
    #ERROR-NR
    #LINE
    #STATUS-CODE
    #PROGRAM
    #LEVEL
    #LEVELI4
END-IF
WRITE #STATUS-CODE
* DECIDE ON FIRST VALUE OF STATUS-CODE
*  ... /* process error
* END-DECIDE
END
```

Einige der oben auf dem Stack abgelegten Informationen entsprechen den Inhalten von Systemvariablen, welche in einem `ON ERROR`-Statement-Block zur Verfügung stehen.

Feldinhalt	**Entsprechende Systemvariable im ON ERROR-Statement-Block**
Natural-Fehlernummer	`*ERROR-NR`
Nummer der Zeile, in welcher der Fehler aufgetreten ist	`*ERROR-LINE`
Name des Natural-Objekts, in dem der Fehler aufgetreten ist	`*PROGRAM`
Nummer der Programmebene, auf der der Fehler aufgetreten ist	`*LEVEL`

Regeln unter Natural Security

Wenn Natural Security installiert ist, gelten zusätzliche Regeln für die Verarbeitung von Fehlern, die beim Anmelden auftreten. Weitere Informationen siehe *Transactions* in der *Natural Security*-Dokumentation.

Funktionalität für die Fehlerverarbeitung

Natural bietet Ihnen umfangreiche Funktionalität, die Sie im Zusammenhang mit der Fehlerverarbeitung verwenden können. Sie können damit

- das Verhalten von Natural bei der Fehlerverarbeitung konfigurieren,
- Informationen über aufgetretene Fehler abrufen,
- Unterstützung bei der Verarbeitung dieser Fehler anfordern,
- Unterstützung bei der Bereinigung von Anwendungsfehlern erhalten.

Diese Funktionalität umfasst spezielle:

- **Profilparameter**
- **Systemvariablen**
- **Terminalkommandos**
- **Systemkommandos**
- **Anwendungsprogrammierschnittstellen (APIs)**

Profilparameter

Folgende Profilparameter beeinflussen das Verhalten von Natural im Fehlerfall:

Profilparameter	Zweck
CPCVERR	Konvertierungsfehler
ETA	Fehlertransaktionsprogramm
ETEOP	Ausgabe eines END TRANSACTION-Statements bei einem End of Program
RCFIND	Handhabung des Response Code 113 beim FIND-Statement
RCGET	Handhabung des Response Code 113 beim GET-Statement
SYNERR	Überwachung von Syntaxfehlern

Systemvariablen

Die folgenden anwendungsbezogenen Systemvariablen können verwendet werden, um einen Fehler zu lokalisieren oder um den Namen des Programms zu erhalten bzw. anzugeben, das die Kontrolle im Fehlerfall erhalten soll:

Systemvariable	Inhalt
*ERROR-LINE	Source-Zeilennummer des Statements, das den Fehler verursacht hat. Siehe **Beispiel 1**.
*ERROR-NR	Fehlernummer des Fehlers, der eine abzuarbeitende ON ERROR-Bedingung verursacht hat.
*ERROR-TA	Name des Programms, das die Kontrolle im Fehlerfall erhalten soll. Siehe **Beispiel 2**.
*LEVEL	Nummer der Ebene des Natural-Objekts, auf der der Fehler aufgetreten ist.
*LIBRARY-ID	Name der Library, in welcher der Benutzer zurzeit angemeldet ist.
*PROGRAM	Name des Natural-Objekts, das zurzeit ausgeführt wird. Siehe **Beispiel 1**.

Beispiel 1:

```
...
    /*
    ON ERROR
      IF *ERROR-NR = 3009 THEN
        WRITE 'LAST TRANSACTION NOT SUCCESSFUL'
            / 'HIT ENTER TO RESTART PROGRAM'
        FETCH 'ONEEX1'
      END-IF
      WRITE 'ERROR' *ERROR-NR 'OCCURRED IN PROGRAM' *PROGRAM
            'AT LINE' *ERROR-LINE
      FETCH 'MENU'
    END-ERROR
    /*
...
```

Beispiel 2:

```
...
    *ERROR-TA := 'ERRORTA1'
    /* from now on, program ERRORTA1 will be invoked
    /* to process application errors
...
    MOVE 'ERRORTA2' TO *ERROR-TA
    /* change error transaction program to ERRORTA2
...
```

Weitere Informationen zu diesen Systemvariablen finden Sie in den entsprechenden Abschnitten der *Systemvariablen*-Dokumentation.

Terminalkommandos

Das folgende Terminalkommando beeinflusst das Verhalten von Natural im Fehlerfall:

Terminalkommando	Zweck
%E=	Fehlerbehandlung ein-/ausschalten

Systemkommandos

Die folgenden Systemkommandos liefern zusätzliche Informationen über eine Fehlersituation bzw. dienen zum Aufrufen von Utilities zur Fehlerbereinigung bei oder zum Protokollieren von Datenbankaufrufen:

Systemkommando	Zweck
LASTMSG	Anzeige von zusätzlichen Informationen zu der zuletzt aufgetretenen Fehlersituation.
TECH	Anzeige von technischen und sonstigen Informationen über Ihre Natural-Session, zum Beispiel Informationen über den zuletzt aufgetretenen Fehler.

Anwendungsprogrammierschnittstellen

Die folgenden Anwendungsprogrammierschnittstellen (APIs) stehen generell zur Verfügung, um zusätzliche Informationen über eine Fehlersituation abzurufen oder um eine Fehlertransaktion einzurichten.

API	Zweck
USR0040N	Art des letzten Fehlers abfragen
USR1016N	Fehlerebene bei Copycode zeigen
USR2001N	Informationen über letzten Fehler auslesen
USR2006N	Ausführliche Informationen zur Meldung holen
USR2007N	Informationen zum Standard-RPC-Server setzen/abrufen
USR2010N	Informationen über Datenbankfehler zeigen
USR2026N	Technische Informationen holen (TECH)
USR2030N	Dynamische Fehler-Teile :1:,... lesen
USR3320N	Fehler-Kurztext auf FNAT bzw. FUSER suchen
USR4214N	Information über Programmebene zeigen

Weitere Informationen siehe *SYSEXT - Natural Application Programming Interfaces* in der *Utilities*-Dokumentation.

46 Bedingte Verarbeitung — Das IF-Statement

- Struktur des IF-Statements 404
- Geschachtelte IF-Statements 406

Mit dem IF-Statement können Sie eine logische Bedingung definieren und Statements angeben, die in Abhängigkeit von dieser logischen Bedingung verarbeitet werden sollen.

Struktur des IF-Statements

Das IF-Statement hat drei Bestandteile: IF, THEN und ELSE.

IF	Mit der IF-Klausel geben Sie eine logische Bedingung an, die erfüllt werden soll.
THEN	Mit der THEN-Klausel geben Sie die Statements an, die ausgeführt werden sollen, wenn diese Bedingung erfüllt wird.
ELSE	Mit der (wahlweise verwendbaren) ELSE-Klausel haben Sie zusätzlich die Möglichkeit, Statements anzugeben, die ausgeführt werden sollen, wenn die Bedingung *nicht* erfüllt wird.

Ein IF-Statement hat also folgende allgemeine Form:

```
IF condition
   THEN execute statement(s)
   ELSE execute other statement(s)
END-IF
```

Anmerkung: Falls Sie wünschen, dass eine bestimmte Verarbeitung nur ausgeführt werden soll, wenn eine IF-Bedingung *nicht* erfüllt wird, können Sie die Klausel THEN IGNORE verwenden, d.h. das IGNORE-Statement bewirkt, dass die IF-Bedingung ignoriert wird, wenn sie erfüllt wird.

Beispiel 1:

```
** Example 'IFX01': IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 CITY
  2 SALARY (1:1)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY CITY STARTING FROM 'C'
 IF SALARY (1) LT 40000 THEN
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
```

```
    DISPLAY NAME BIRTH (EM=YYYY-MM-DD) SALARY (1)
  END-IF
END-READ
END
```

Der IF-Statement-Block im obigen Programm bewirkt folgende bedingte Verarbeitung:

- Wenn (IF) das Gehalt weniger als 40000 beträgt, dann (THEN) soll das WRITE-Statement ausgeführt werden;
- andernfalls (ELSE), d.h. wenn das Gehalt 40000 und mehr beträgt, soll das DISPLAY-Statement ausgeführt werden.

Ausgabe des Programms IFX01:

```
        NAME             DATE      ANNUAL
                          OF       SALARY
                         BIRTH
-------------------- ---------- ----------

***** KEEN                                            SALARY LT 40000
***** FORRESTER                                       SALARY LT 40000
***** JONES                                           SALARY LT 40000
***** MELKANOFF                                       SALARY LT 40000
DAVENPORT            1948-12-25      42000
GEORGES              1949-10-26     182800
***** FULLERTON                                       SALARY LT 40000
```

Beispiel 2:

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
 IF #INCOME > 40000
```

```
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
    MOVE 'CATALOG I'          TO #TEXT
  END-IF
  /*
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `IFX03`:

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME              SALARY
                          BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                         ----------
                INCOME:       56000   CATALOGS I AND II
                ===================

RICHMOND                       9150
                                  0
                         ----------
                INCOME:        9150   CATALOG I
                ===================

MONKTON                       13500
                                600
                         ----------
                INCOME:       14100   CATALOG I
                ===================
```

Geschachtelte IF-Statements

Es ist möglich, mehrere `IF`-Statements ineinander zu verschachteln, zum Beispiel, indem Sie die Ausführung einer `THEN`-Klausel durch ein weiteres, in der `THEN`-Klausel angegebenes `IF`-Statement von einer zusätzlichen Bedingung abhängig machen.

Beispiel:

```
** Example 'IFX02': IF (two IF statements nested)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY (1:1)
  2 BIRTH
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
*
1 #BIRTH   (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
*
LIMIT 20
FND1. FIND MYVIEW WITH CITY = 'BOSTON'
                  SORTED BY NAME
 IF SALARY (1) LESS THAN 20000
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 20000'
  ELSE
    IF BIRTH GT #BIRTH
      FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
        DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
                        SALARY (1) MAKE (AL=8 IS=OFF)
      END-FIND
    END-IF
  END-IF
  SKIP 1
END-FIND
END
```

Ausgabe des Programms `IFX02`:

```
        NAME            DATE      ANNUAL     MAKE
                         OF       SALARY
                       BIRTH
-------------------- ---------- ---------- --------

***** COHEN                                          SALARY LT 20000

CREMER               1972-12-14      20000 FORD

***** FLEMING                                        SALARY LT 20000
```

```
PERREAULT            1950-05-12      30500 CHRYSLER

***** SHAW                                                  SALARY LT 20000

STANWOOD             1946-09-08      31000 CHRYSLER
                                           FORD
```

47 Schleifenverarbeitung

- Verwendung von Verarbeitungsschleifen 410
- Schleifendurchläufe bei Datenbankzugriffen begrenzen 410
- Durchläufe bei Nicht-Datenbankzugriffsschleifen begrenzen — das REPEAT-Statement 412
- Beispiel für Verarbeitungsschleife mit REPEAT-Statement 413
- Verarbeitungsschleife verlassen — das ESCAPE-Statement 414
- Schleifen innerhalb von Schleifen 414
- Beispiel für geschachtelte FIND-Statements 415
- Statements innerhalb eines Programms referenzieren 416
- Beispiel für das Referenzieren mit Zeilennummern 418
- Beispiel mit Statement-Labels 418

Eine Verarbeitungsschleife ist eine Gruppe von Statements, deren Ausführung so oft wiederholt wird, bis eine bestimmte Bedingung erfüllt ist, oder solange eine bestimmte Bedingung gegeben ist.

Verwendung von Verarbeitungsschleifen

Verarbeitungsschleifen lassen sich in Datenbankschleifen und Nicht-Datenbankschleifen unterteilen:

- **Datenbankschleifen**
  werden von Natural automatisch erzeugt, um die Daten, die mit einem `READ`-, `FIND`- oder `HISTOGRAM`-Statement von einer Datenbank gelesen werden, zu verarbeiten. Diese Statements sind im Kapitel ***Datenbankzugriffe*** beschrieben.
- **Nicht-Datenbankschleifen**
  (d.h. Schleifen ohne Datenbankzugriff) werden mit folgenden Statements erzeugt: `REPEAT`, `FOR`, `CALL FILE`, `CALL LOOP`, `SORT` und `READ WORK FILE`.

Es können mehrere Verarbeitungsschleifen gleichzeitig aktiv sein. In einer gerade aktiven, d.h. noch nicht abgeschlossenen Schleife können weitere Schleifen eingebettet werden.

Jede Verarbeitungsschleife muss durch ein entsprechendes `END-...`-Statement beendet werden (z.B. `END-REPEAT`, `END-FOR` usw.).

Das `SORT`-Statement, mit dem das Sortierprogramm des Betriebssystems aufgerufen wird, beendet alle aktiven Schleifen und löst eine neue Schleife aus.

Schleifendurchläufe bei Datenbankzugriffen begrenzen

Die folgenden Themen werden behandelt:

- Möglichkeiten der Begrenzung von Datenbankschleifen
- LT-Session-Parameter
- LIMIT-Statement
- Limit-Notation

- Priorität der Limit-Einstellungen

Möglichkeiten der Begrenzung von Datenbankschleifen

Bei den Statements READ, FIND oder HISTOGRAM haben Sie drei Möglichkeiten, die Anzahl, wie oft eine Verarbeitungsschleife durchlaufen werden soll, zu begrenzen:

- mit dem Session-Parameter LT
- mit einem LIMIT-Statement
- oder mit einer **Limit-Notation** im READ-/FIND-/HISTOGRAM-Statement selbst.

LT-Session-Parameter

Mit dem Systemkommando GLOBALS können Sie den Session-Parameter LT angeben, der die Anzahl der Datensätze, die in einer Datenbank-Verarbeitungsschleife gelesen werden sollen, begrenzt.

Beispiel:

```
GLOBALS LT=100
```

Dieses Limit gilt für alle READ-, FIND- und HISTOGRAM-Schleifen in der gesamten Session.

LIMIT-Statement

In einem Programm können Sie die Anzahl der Datensätze, die in einer Datenbank-Verarbeitungsschleife gelesen werden sollen, mit einem LIMIT-Statement begrenzen.

Beispiel:

```
LIMIT 100
```

Das LIMIT-Statement gilt für alle nachfolgenden READ-, FIND- oder HISTOGRAM-Schleifen im Programm, es sein denn, es wird durch ein anderes LIMIT-Statement oder eine Limit-Notation außer Kraft gesetzt.

Limit-Notation

In einem READ-, FIND- oder HISTOGRAM-Statement selbst können Sie die Anzahl der Datensätze, die gelesen werden sollen, in Klammern unmittelbar hinter dem Statement-Namen angeben.

Beispiel:

```
READ (10) VIEWXYZ BY NAME
```

Diese Limit-Notation hat Vorrang vor allen anderen Limits, gilt aber nur für das betreffende Statement.

Priorität der Limit-Einstellungen

Wenn das mit dem `LT`-Parameter angegebene Limit kleiner ist als ein mit einem `LIMIT`-Statement oder einer Limit-Notation angegebenes, dann hat das `LT`-Limit Vorrang vor diesen anderen Limits.

Durchläufe bei Nicht-Datenbankzugriffsschleifen begrenzen — das REPEAT-Statement

Anfang und Ende von Verarbeitungsschleifen, die keinen Datenbankzugriff beinhalten, basieren auf einer logischen oder sonstwie die Schleife begrenzenden Bedingung. Sie werden mit einem der folgenden Statements erzeugt: `REPEAT`, `FOR`, `CALL FILE`, `CALL LOOP`, `SORT` und `READ WORK FILE`.

Stellvertretend für Nicht-Datenbankschleifen-Statements wird hier das Statement `REPEAT` behandelt.

Mit dem `REPEAT`-Statement geben Sie ein oder mehrere Statements an, die wiederholt ausgeführt werden sollen. Außerdem können Sie eine logische Bedingung angeben, so dass die Statements nur ausgeführt werden, solange oder bis diese Bedingung erfüllt ist. Die Bedingung geben Sie in einer `UNTIL`-Klausel oder in einer `WHILE`-Klausel an:

- Bei einer `UNTIL`-Klausel wird die Schleife so oft ausgeführt, bis (`UNTIL`) die logische Bedingung erfüllt ist, d.h. die Schleife wird beendet, sobald der in der Bedingung angegebene Zustand erreicht ist.
- Bei einer `WHILE`-Klausel wird die `REPEAT`-Schleife ausgeführt, während (`WHILE`) der in der Bedingung angegebene Zustand besteht, d.h. die Schleife wird beendet, sobald die Bedingung nicht mehr erfüllt wird.

Wenn Sie *keine* logische Bedingung angeben, muss die `REPEAT`-Schleife mit einem der folgenden Statements verlassen werden:

- `ESCAPE` (siehe **nächsten Abschnitt**) beendet die Verarbeitung der Schleife und setzt die Verarbeitung außerhalb der Schleife fort.
- `STOP` bricht die Ausführung der gesamten Natural-Anwendung ab.
- `TERMINATE` bricht die Ausführung der Natural-Anwendung ab und beendet die Natural-Session.

Beispiel für Verarbeitungsschleife mit REPEAT-Statement

```
** Example 'REPEAX01': REPEAT
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1:1)
*
1 #PAY1    (N8)
END-DEFINE
*
READ (5) MYVIEW BY NAME WHERE SALARY (1) = 30000 THRU 39999
  MOVE SALARY (1) TO #PAY1
  /*
  REPEAT WHILE #PAY1 LT 40000
    MULTIPLY #PAY1 BY 1.1
    DISPLAY NAME (IS=ON) SALARY (1)(IS=ON) #PAY1
  END-REPEAT
  /*
  SKIP 1
END-READ
END
```

Ausgabe des Programms `REPEAX01`:

```
Page      1                                               04-11-11  14:15:54

        NAME           ANNUAL     #PAY1
                       SALARY
-------------------- ---------- ---------

ADKINSON                  34500     37950
                                    41745

                          33500     36850
                                    40535

                          36000     39600
                                    43560

AFANASSIEV                37000     40700

ALEXANDER                 34500     37950
                                    41745
```

Verarbeitungsschleife verlassen — das ESCAPE-Statement

Mit dem `ESCAPE`-Statement können Sie die Ausführung einer Verarbeitungsschleife abbrechen, und zwar aufgrund einer logischen Bedingung.

Das `ESCAPE`-Statement kann Teil eines `IF`-Statements sein oder an eines der Statements `AT END OF DATA`, `AT END OF PAGE` oder `AT BREAK` geknüpft sein; es kann aber auch als eigenständiges Statement in Ausführung der einer Verarbeitungsschleife zugrundeliegenden logischen Bedingungen stehen.

Mit dem `ESCAPE`-Statement haben Sie die Optionen `TOP` und `BOTTOM`, mit denen Sie festlegen, wo die Verarbeitung fortgesetzt werden soll, nachdem die Schleife mit `ESCAPE` verlassen wurde:

- Bei `ESCAPE TOP` wird die Verarbeitung am Anfang der Schleife, d.h. mit dem nächsten Schleifendurchlauf, fortgesetzt.
- Bei `ESCAPE BOTTOM` wird die Verarbeitung mit dem ersten Statement, das nach der Schleife kommt, fortgesetzt.

Sie können innerhalb einer Verarbeitungsschleife auch mehrere `ESCAPE`-Statements angeben.

Weitere Informationen und Beispiele zum `ESCAPE`-Statement finden Sie in der *Statements*-Dokumentation.

Schleifen innerhalb von Schleifen

Mit Natural haben Sie die Möglichkeit, Schleifen innerhalb von Schleifen auszulösen und so eine ganze „Hierarchie“ ineinander verschachtelter Schleifenkonstruktionen aufzubauen. Sind mehrere Datenbankzugriffsschleifen ineinander verschachtelt, so durchläuft jeder gelesene Datensatz, der die Auswahlkriterien erfüllt, nacheinander die einzelnen Schleifen, bevor der nächste Datensatz verarbeitet wird.

Mehrere Datenbankzugriffs- und Nicht-Datenbankzugriffsschleifen können ineinander verschachtelt werden. Verarbeitungsschleifen können auch Teil einer bedingten Verarbeitung sein.

Beispiel für geschachtelte FIND-Statements

Das folgende Programm zeigt eine Hierarchie zweier Verarbeitungsschleifen, wobei sich eine FIND-Schleife innerhalb einer anderen FIND-Schleife befindet.

```
** Example 'FINDX06': FIND (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 PERSONNEL-ID
1 VEH-VIEW VIEW OF VEHICLES
  2 MAKE
  2 PERSONNEL-ID
END-DEFINE
*
FND1. FIND EMPLOY-VIEW WITH CITY = 'NEW YORK' OR = 'BEVERLEY HILLS'
  FIND (1) VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
    DISPLAY NOTITLE NAME CITY MAKE
  END-FIND
END-FIND
END
```

Das obige Programm liest Daten von mehreren Dateien. Die äußere FIND-Schleife wählt von der EMPLOYEES-Datei alle Personen aus, die in New York oder Beverley Hills wohnen. Für jeden in der äußeren Schleife ausgewählten Datensatz wird die innere FIND-Schleife durchlaufen, in der die Fahrzeugdaten der betreffenden Personen von der VEHICLES-Datei gelesen werden.

Ausgabe des Programms FINDX06:

```
        NAME                 CITY                 MAKE
-------------------- -------------------- --------------------

RUBIN                NEW YORK             FORD
OLLE                 BEVERLEY HILLS       GENERAL MOTORS
WALLACE              NEW YORK             MAZDA
JONES                BEVERLEY HILLS       FORD
SPEISER              BEVERLEY HILLS       GENERAL MOTORS
```

Statements innerhalb eines Programms referenzieren

Statement-Referenzierung dient dazu

- in einem Programm auf ein vorhergehendes Statement zu verweisen (d.h. dieses Statement zu „referenzieren"), um eine Verarbeitung für einen bestimmten Bereich von Daten auszuführen,
- Naturals **Standard-Referenzierung** (die bei jedem betroffenen Statement in der Dokumentation beschrieben ist) aufzuheben
- oder zu Programmdokumentationszwecken.

Sie können jedes Natural-Statement referenzieren, das eine Verarbeitungsschleife initiiert und/oder auf Datenelemente in einer Datenbank zugreift:

- `READ`
- `FIND`
- `HISTOGRAM`
- `SORT`
- `REPEAT`
- `FOR`

Enthält ein Programm mehrere Verarbeitungsschleifen, so kann man ein bestimmtes Datenbankfeld eindeutig identifizieren, indem man das Statement referenziert, welches zuerst auf das entsprechende Feld in der Datenbank zugriff.

Welche Felder bei welchem Statement referenziert werden dürfen, ersehen Sie in der Statements-Dokumentation in den *Operandentabellen* der einzelnen Statements aus der Spalte *Referenzierung erlaubt*. Siehe auch ***Benutzervariablen***, *Datenbankfelder mit der* `(r)`*-Notation referenzieren*.

Außerdem kann eine Referenzierungsnotation in einigen Statements angegeben werden, z.B. bei:

- `AT START OF DATA`
- `AT END OF DATA`
- `AT BREAK`
- `ESCAPE BOTTOM`

Normalerweise bezieht sich bei einem `AT START OF DATA`-, `AT END OF DATA`- oder `AT BREAK`-Statement die schleifenbeendende Gruppenwechsel-Bedingung auf die jeweils äußerste aktive `READ`-, `FIND`-, `HISTOGRAM`-, `SORT`- oder `READ WORK FILE`-Schleife. Mit einer Referenzierungsnotation können Sie die Bedingung auf eine beliebige andere aktive Schleife beziehen.

Wenn Sie bei einem `ESCAPE BOTTOM`-Statement ein Statement referenzieren, wird die Verarbeitung unmittelbar nach der durch das referenzierte Statement identifizierten Schleife fortgesetzt.

Zur Statement-Referenzierung können Sie entweder ein sogenanntes *Statement-Label* oder die *Sourcecode-Zeilennummer* verwenden.

- **Statement-Label**
  Ein Statement-Label ist eine Zeichenkette, deren letztes Zeichen ein Punkt (.) sein muss. Der Punkt identifiziert die Zeichenkette als Label.

  Ein Statement, das referenziert werden soll, wird mit einem Label markiert, indem das Label an den Anfang der Zeile gestellt wird, in der das Statement steht, zum Beispiel:

```
0030 ...
0040 READ1. READ VIEWXYZ BY NAME
0050 ...
```

  In dem Statement, das das markierte Statement referenziert, wird das Label in Klammern an der in der Statement-Syntax dafür vorgesehenen Stelle (siehe Syntaxdiagramme in der *Statements*-Dokumentation) eingefügt, zum Beispiel:

```
AT BREAK (READ1.) OF NAME
```

- **Sourcecode-Zeilennummern**
  Wenn Sie Sourcecode-Zeilennummern zur Referenzierung verwenden, müssen Sie diese immer vierstellig (vorangestellte Nullen dürfen nicht weggelassen werden) und in Klammern angeben, zum Beispiel:

```
AT BREAK (0040) OF NAME
```

  Bezieht sich in einem Statement ein bestimmtes Feld auf ein vorhergegangenes Statement, so wird das Label bzw. die Zeilennummer in Klammern hinter dem jeweiligen Feldnamen angegeben, zum Beispiel:

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

Sourcecode-Zeilennummern und Statement-Labels können wahlweise verwendet werden.

Siehe auch ***Benutzervariablen***, ***Datenbankfelder mit der (r)- Notation referenzieren***.

Beispiel für das Referenzieren mit Zeilennummern

Das folgende Programm verwendet Sourcecode-Zeilennummern (vierstellige Ziffern in Klammern) zur Referenzierung.

In diesem Beispiel beziehen sich die Zeilennummern auf Statements, die aufgrund der Programmstruktur ohnehin, auch ohne explizite Referenzierung, referenziert worden wären.

```
0010 ** Example 'LABELX01': Labels for READ and FIND loops (line numbers)
0020 ************************************************************************
0030 DEFINE DATA LOCAL
0040 1 MYVIEW1 VIEW OF EMPLOYEES
0050   2 NAME
0060  2 FIRST-NAME
0070  2 PERSONNEL-ID
0080 1 MYVIEW2 VIEW OF VEHICLES
0090  2 PERSONNEL-ID
0100  2 MAKE
0110 END-DEFINE
0120 *
0130 LIMIT 15
0140 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0150  FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0140)
0160    IF NO RECORDS FOUND
0170      MOVE '***NO CAR***' TO MAKE
0180    END-NOREC
0190    DISPLAY NOTITLE NAME       (0140) (IS=ON)
0200                    FIRST-NAME (0140) (IS=ON)
0210                    MAKE       (0150)
0220  END-FIND /* (0150)
0230 END-READ   /* (0140)
0240 END
```

Beispiel mit Statement-Labels

Das folgende Beispiel zeigt die Verwendung von Statement-Labels.

Es ist mit dem vorigen Beispielprogramm identisch bis auf die Tatsache, dass zur Referenzierung der Statements Labels anstelle von Zeilennummern verwendet werden.

```
** Example 'LABELX02': Labels for READ and FIND loops (user labels)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW1 VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
  FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '***NO CAR***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE NAME       (RD.) (IS=ON)
                    FIRST-NAME (RD.) (IS=ON)
                    MAKE       (FD.)
  END-FIND /* (FD.)
END-READ   /* (RD.)
END
```

Beide Programme erzeugen folgende Ausgabe:

```
        NAME              FIRST-NAME              MAKE
-------------------- -------------------- --------------------

JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
                                          MG
                     EDWARD               GENERAL MOTORS
                     LAUREL               GENERAL MOTORS
                     KEVIN                DATSUN
                     GREGORY              FORD
JOPER                MANFRED              ***NO CAR***
JOUSSELIN            DANIEL               RENAULT
JUBE                 GABRIEL              ***NO CAR***
JUNG                 ERNST                ***NO CAR***
JUNKIN               JEREMY               ***NO CAR***
KAISER               REINER               ***NO CAR***
KANT                 HEIKE                ***NO CAR***
```

48 Gruppenwechsel

- Verwendung von Gruppenwechseln 422
- AT BREAK-Statement 422
- Automatische Gruppenwechsel-Verarbeitung 428
- Beispiel für Systemfunktionen in einem AT BREAK-Statement 430
- Weiteres Beispiel für AT BREAK-Statement 431
- BEFORE BREAK PROCESSING-Statement 431
- Beispiel für BEFORE BREAK PROCESSING-Statement 431
- Programmabhängige Gruppenwechsel-Verarbeitung — das PERFORM BREAK PROCESSING-Statement 432
- Beispiel für PERFORM BREAK PROCESSING-Statement 434

Dieses Kapitel beschreibt, wie die Ausführung eines Statements von einem Gruppenwechsel abhängig gemacht werden kann, und wie Gruppenwechsel für die Auswertung von Natural-Systemfunktionen benutzt werden können.

Verwendung von Gruppenwechseln

Ein Gruppenwechsel (Break) findet statt, wenn der Wert eines Kontrollfeldes sich ändert.

Die Ausführung von Statements kann von einem solchen Gruppenwechsel abhängig gemacht werden.

Ein Gruppenwechsel kann auch zur Auswertung von Natural-Systemfunktionen verwendet werden.

Systemfunktionen werden im Abschnitt ***Systemvariablen und Systemfunktionen*** behandelt. Genauere Beschreibungen der verfügbaren Systemfunktionen System finden Sie in der *Systemfunktionen*-Dokumentation.

AT BREAK-Statement

Mit dem Statement `AT BREAK` können Sie eine Verarbeitung angeben, die immer dann ausgeführt werden soll, wenn ein Gruppenwechsel erfolgt, d.h. wenn der Wert eines Kontrollfeldes, das Sie im `AT BREAK`-Statement angeben, sich ändert. Als Kontrollfeld können Sie ein Datenbankfeld oder eine Benutzervariable verwenden.

In diesem Abschnitt werden folgende Themen behandelt:

- Gruppenwechsel basierend auf einem Datenbankfeld
- Gruppenwechsel basierend auf einer Benutzervariablen
- Gruppenwechsel auf mehreren Ebenen

Gruppenwechsel basierend auf einem Datenbankfeld

Das Feld, welches als Kontrollfeld in einem `AT BREAK`-Statement angegeben wird, ist üblicherweise ein Datenbankfeld.

Beispiel:

```
...
AT BREAK OF DEPT
  statements
END-BREAK
...
```

In diesem Beispiel ist das Datenbankfeld `DEPT` das Kontrollfeld; wechselt der Wert des Feldes, beispielsweise von `SALE01` auf `SALE02`, würde dies die Ausführung der im `AT BREAK`-Statement angegebenen *Statements* auslösen.

Es ist auch möglich, statt eines ganzen Feldes nur einen Teil eines Feldes als Kontrollfeld zu nehmen. Mit der Notation `/n/` können Sie festlegen, dass nur die ersten `/n/` Stellen eines Feldes auf einen Wertwechsel überprüft werden sollen.

Beispiel:

```
...
AT BREAK OF DEPT /4/
  statements
END-BREAK
...
```

In diesem Beispiel würden die angegebenen *Statements* nur ausgeführt, wenn sich der Wert der ersten 4 Stellen des Feldes `DEPT` ändern würde, beispielsweise von `SALE` auf `TECH`; ein Wechsel von `SALE01` auf `SALE02` hingegen würde ignoriert und der `AT BREAK`-Block nicht ausgeführt werden.

Beispiel:

```
** Example 'ATBREX01': AT BREAK OF (with database field)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  DISPLAY CITY (AL=9) NAME 'POSITION' JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT BREAK OF CITY
    WRITE /   OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^)
```

```
         5X 'AVERAGE:' T*SALARY AVER(SALARY(1)) //
             COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 'TOTAL (ALL RECORDS):' T*SALARY(1) TOTAL(SALARY(1))
  END-ENDDATA
END-READ
END
```

Im obigen Programm wird das erste WRITE-Statement ausgeführt, wenn der Wert des Feldes CITY sich ändert.

Im AT BREAK-Statement werden die Systemfunktionen OLD, AVER und COUNT ausgewertet (und in dem WRITE-Statement ausgegeben).

In dem AT END OF DATA-Statement wird die Systemfunktion TOTAL ausgewertet.

Das Programm ATBREX01 erzeugt folgende Ausgabe:

```
Page      1                                                04-12-14  14:07:26

  CITY            NAME                 POSITION            SALARY
--------- -------------------- ------------------------- ----------

AIKEN     SENKO                PROGRAMMER                     31500

A I K E N                 AVERAGE:                            31500

       1 RECORDS FOUND

ALBUQUERQ HAMMOND              SECRETARY                      22000
ALBUQUERQ ROLLING              MANAGER                        34000
ALBUQUERQ FREEMAN              MANAGER                        34000
ALBUQUERQ LINCOLN              ANALYST                        41000

A L B U Q U E R Q U E     AVERAGE:                            32750

       4 RECORDS FOUND

TOTAL (ALL RECORDS):                                         162500
```

Gruppenwechsel basierend auf einer Benutzervariablen

Auch eine **Benutzervariable** kann als Kontrollfeld in einem `AT BREAK`-Statement verwendet werden.

Im folgenden Programm wird die Benutzervariable `#LOCATION` als Kontrollfeld verwendet.

```
** Example 'ATBREX02': AT BREAK OF  (with user-defined variable and
**                     in conjunction with BEFORE BREAK PROCESSING)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
*
1 #LOCATION (A20)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  BEFORE BREAK PROCESSING
    COMPRESS CITY 'USA' INTO #LOCATION
  END-BEFORE
  DISPLAY #LOCATION 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  /*
  AT BREAK OF #LOCATION
    SKIP 1
  END-BREAK
END-READ
END
```

Ausgabe des Programms `ATBREX02`:

```
Page      1                                            04-12-14  14:08:36

     #LOCATION               POSITION           SALARY
-------------------- ------------------------- ----------

AIKEN USA            PROGRAMMER                     31500

ALBUQUERQUE USA      SECRETARY                      22000
ALBUQUERQUE USA      MANAGER                        34000
ALBUQUERQUE USA      MANAGER                        34000
ALBUQUERQUE USA      ANALYST                        41000
```

Gruppenwechsel auf mehreren Ebenen

Mit der Notation /*n*/ können Sie, wie **oben** erläutert, den Teil eines Feldes zum Kontrollfeld eines Gruppenwechsels machen. Sie können auch mehrere AT BREAK-Statements miteinander kombinieren, wobei bei einem Gruppenwechsel ein ganzes Feld und bei einem anderen ein Teil dieses Feldes Kontrollfeld ist.

In diesem Fall muss der übergeordnete Gruppenwechsel (ganzes Feld) vor dem untergeordneten (Teil des Feldes) angegeben werden, d.h. im ersten AT BREAK-Statement muss das ganze Feld, im zweiten das Teilfeld als Kontrollfeld angegeben werden.

Das folgende Beispielprogramm zeigt dies anhand des Feldes DEPT und den ersten 4 Stellen dieses Feldes (DEPT /4/).

```
** Example 'ATBREX03': AT BREAK OF (two statements in combination)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
READ MYVIEW BY DEPT STARTING FROM 'SALE40' ENDING AT 'TECH10'
     WHERE SALARY(1) GT 47000 AND CURR-CODE(1) = 'USD'
  /*
  AT BREAK OF DEPT
    WRITE '*** LOWEST BREAK LEVEL ***' /
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE '*** HIGHEST BREAK LEVEL ***'
  END-BREAK
  /*
  DISPLAY DEPT NAME 'POSITION' JOB-TITLE
END-READ
END
```

Ausgabe des Programms ATBREX03:

```
Page      1                                               04-12-14  14:09:20

DEPARTMENT         NAME                 POSITION
   CODE
---------- -------------------- -------------------------

TECH05     HERZOG               MANAGER
TECH05     LAWLER               MANAGER
TECH05     MEYER                MANAGER
*** LOWEST BREAK LEVEL ***

TECH10     DEKKER               DBA
*** LOWEST BREAK LEVEL ***

*** HIGHEST BREAK LEVEL ***
```

Im folgenden Programm wird jedesmal, wenn sich der Wert des Feldes `DEPT` ändert, eine Leerzeile ausgegeben; und jedesmal, wenn sich der Wert in den ersten 4 Stellen von DEPT ändert, wird über die Systemfunktion `COUNT` die Anzahl der verarbeiteten Datensätze ermittelt.

```
** Example 'ATBREX04': AT BREAK OF (two statements in combination)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 DEPT
  2 REDEFINE DEPT
    3 #GENDEP (A4)
  2 NAME
  2 SALARY    (1)
END-DEFINE
*
WRITE TITLE '** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **' /
LIMIT 9
READ MYVIEW BY DEPT FROM 'A' WHERE SALARY(1) > 30000
  DISPLAY 'DEPT' DEPT NAME 'SALARY' SALARY(1)
  /*
  AT BREAK OF DEPT
    SKIP 1
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE COUNT(SALARY(1)) 'RECORDS FOUND IN:' OLD(#GENDEP) /
  END-BREAK
END-READ
END
```

Ausgabe des Programms `ATBREX04`:

```
           ** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **

 DEPT          NAME           SALARY
------ -------------------- ----------

ADMA01 JENSEN                   180000
ADMA01 PETERSEN                 105000
ADMA01 MORTENSEN                320000
ADMA01 MADSEN                   149000
ADMA01 BUHL                     642000

ADMA02 HERMANSEN                391500
ADMA02 PLOUG                    162900
ADMA02 HANSEN                   234000

       8 RECORDS FOUND IN: ADMA

COMP01 HEURTEBISE               168800

       1 RECORDS FOUND IN: COMP
```

Automatische Gruppenwechsel-Verarbeitung

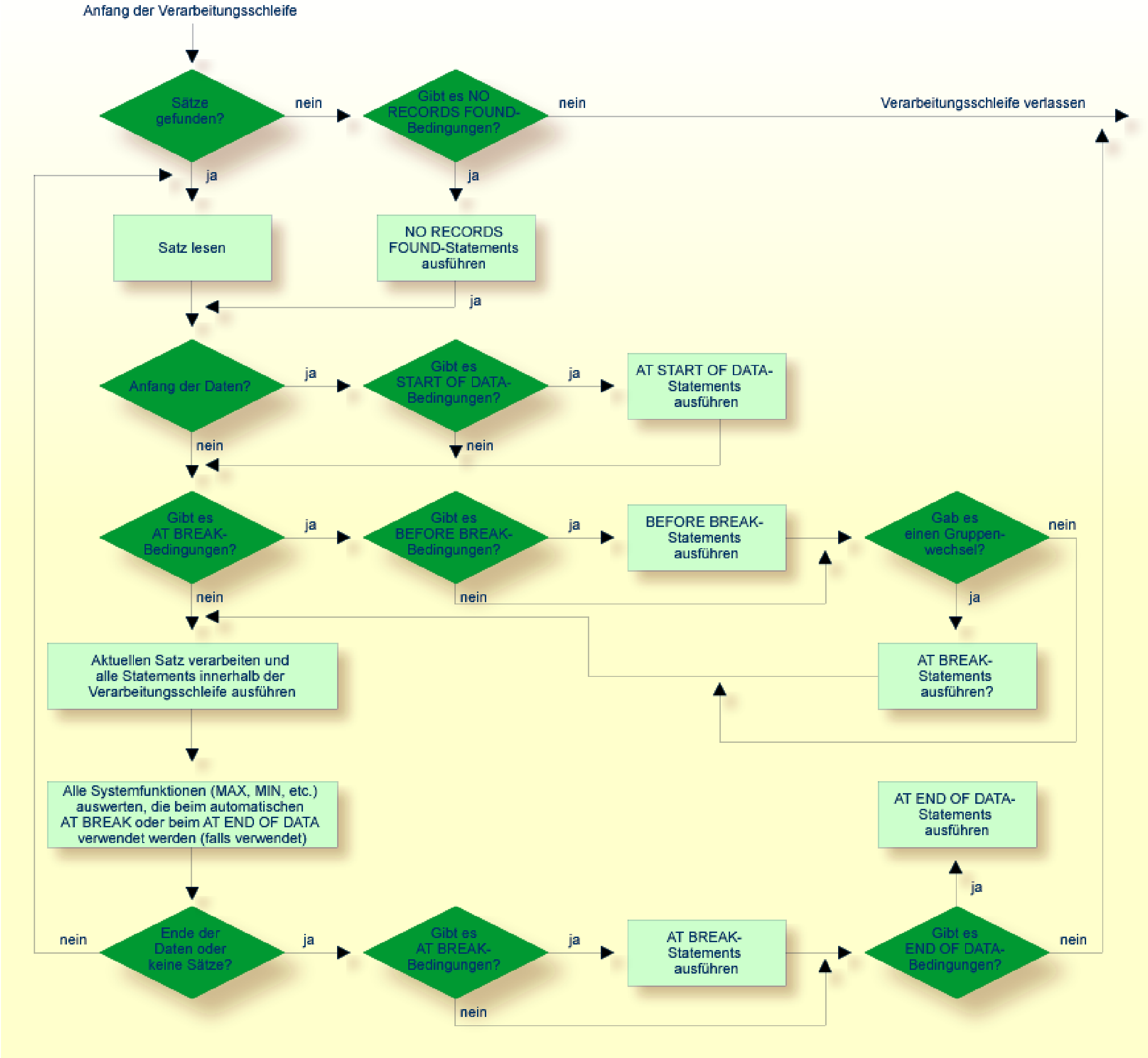
Automatische Gruppenwechsel-Verarbeitung ist für eine Verarbeitungsschleife aktiv, die ein AT BREAK-Statement enthält. Dies gilt für die folgenden Statements:

- FIND
- READ
- HISTOGRAM
- SORT
- READ WORK FILE

Hierbei wird der Wert des im AT BREAK-Statement angegebenen Kontrollfeldes nur bei den Datensätzen überprüft, die die WITH- und WHERE-Auswahlkriterien der Verarbeitungsschleife erfüllen.

Natural-Systemfunktionen (AVER, MAX, MIN usw.) werden für jeden Datensatz ausgewertet, nachdem alle in der Verarbeitungsschleife enthaltenen Statements ausgeführt worden sind. Datensätze, die aufgrund des WHERE-Kriteriums nicht verarbeitet werden, werden bei der Auswertung der Systemfunktionen nicht berücksichtigt.

Die Abbildung auf der folgenden Seite veranschaulicht den Verarbeitungsablauf eines automatischen Gruppenwechsels.



Beispiel für Systemfunktionen in einem AT BREAK-Statement

Das folgende Beispiel zeigt die Verwendung der Natural-Systemfunktionen OLD, MIN, AVER, MAX, SUM und COUNT in einem AT BREAK-Statement (und der Systemfunktion TOTAL in einem AT END OF DATA-Statement).

```
** Example 'ATBREX05': AT BREAK OF (with system functions)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
LIMIT 3
READ MYVIEW BY CITY = 'SALT LAKE CITY'
  DISPLAY NOTITLE CITY NAME 'SALARY' SALARY(1) 'CURRENCY' CURR-CODE(1)
  /*
  AT BREAK OF CITY
    WRITE /  OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X)
      31T ' - MINIMUM:' MIN(SALARY(1))  CURR-CODE(1) /
      31T ' - AVERAGE:' AVER(SALARY(1)) CURR-CODE(1) /
      31T ' - MAXIMUM:' MAX(SALARY(1))  CURR-CODE(1) /
      31T ' -     SUM:' SUM(SALARY(1))  CURR-CODE(1) /
      33T COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 22T 'TOTAL (ALL RECORDS):'
              T*SALARY TOTAL(SALARY(1)) CURR-CODE(1)
  END-ENDDATA
END-READ
END
```

Ausgabe des Programms ATBREX05:

```
        CITY                 NAME           SALARY   CURRENCY
-------------------- -------------------- ---------- --------

SALT LAKE CITY       ANDERSON                  50000 USD
SALT LAKE CITY       SAMUELSON                 24000 USD

S A L T   L A K E   C I T Y    - MINIMUM:      24000 USD
                               - AVERAGE:      37000 USD
```

```
                               - MAXIMUM:      50000 USD
                               -     SUM:      74000 USD
                                       2 RECORDS FOUND

SAN DIEGO           GEE                        60000 USD

S A N   D I E G O              - MINIMUM:      60000 USD
                               - AVERAGE:      60000 USD
                               - MAXIMUM:      60000 USD
                               -     SUM:      60000 USD
                                       1 RECORDS FOUND

                     TOTAL (ALL RECORDS):     134000 USD
```

Weiteres Beispiel für AT BREAK-Statement

Siehe folgendes Beispielprogramm:

- *ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)*

BEFORE BREAK PROCESSING-Statement

Mit dem Statement `BEFORE BREAK PROCESSING` können Sie Statements angeben, die unmittelbar vor einem Gruppenwechsel ausgeführt werden sollen, d.h. bevor der Wert des Kontrollfeldes geprüft wird, bevor die Statements im `AT BREAK`-Block ausgeführt werden und bevor Natural-Systemfunktionen ausgewertet werden.

Beispiel für BEFORE BREAK PROCESSING-Statement

```
** Example 'BEFORX01': BEFORE BREAK PROCESSING
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
*
1 #INCOME  (P11)
END-DEFINE
*
```

```
LIMIT 5
READ MYVIEW BY NAME FROM 'B'
 BEFORE BREAK PROCESSING
    COMPUTE #INCOME = SALARY(1) + BONUS(1,1)
  END-BEFORE
  /*
  DISPLAY NOTITLE NAME FIRST-NAME (AL=10)
                 'ANNUAL/INCOME' #INCOME 'SALARY' SALARY(1) (LC==) /
                 '+ BONUS' BONUS(1,1) (IC=+)
  AT BREAK OF #INCOME
    WRITE T*#INCOME '-'(24)
  END-BREAK
END-READ
END
```

Ausgabe des Programms `BEFORX01`:

```
        NAME         FIRST-NAME    ANNUAL      SALARY
                                   INCOME      + BONUS
-------------------- ---------- ------------ -----------

BACHMANN             HANS              56800 =     52800
                                                   +4000
                                ------------------------
BAECKER              JOHANNES          81000 =     74400
                                                   +6600
                                ------------------------
BAECKER              KARL              52650 =     48600
                                                   +4050
                                ------------------------
BAGAZJA              MARJAN           152700 =    129700
                                                  +23000
                                ------------------------
BAILLET              PATRICK          198500 =    188000
                                                  +10500
                                ------------------------
```

Programmabhängige Gruppenwechsel-Verarbeitung — das PERFORM BREAK PROCESSING-Statement

Bei automatischer Gruppenwechsel-Verarbeitung werden die im AT BREAK-Block angegebenen Statements jedesmal ausgeführt, wenn sich der Wert des angegebenen Kontrollfeldes ändert, und zwar unabhängig von der Position des `AT BREAK`-Statements in der Verarbeitungsschleife.

Mit einem `PERFORM BREAK PROCESSING`-Statement können Sie selbst festlegen, wo in einer Verarbeitungsschleife eine Gruppenwechsel-Verarbeitung ausgeführt werden soll. Das `PERFORM BREAK`

PROCESSING-Statement wird dann ausgeführt, wenn es im Verarbeitungsablauf des Programms angetroffen wird.

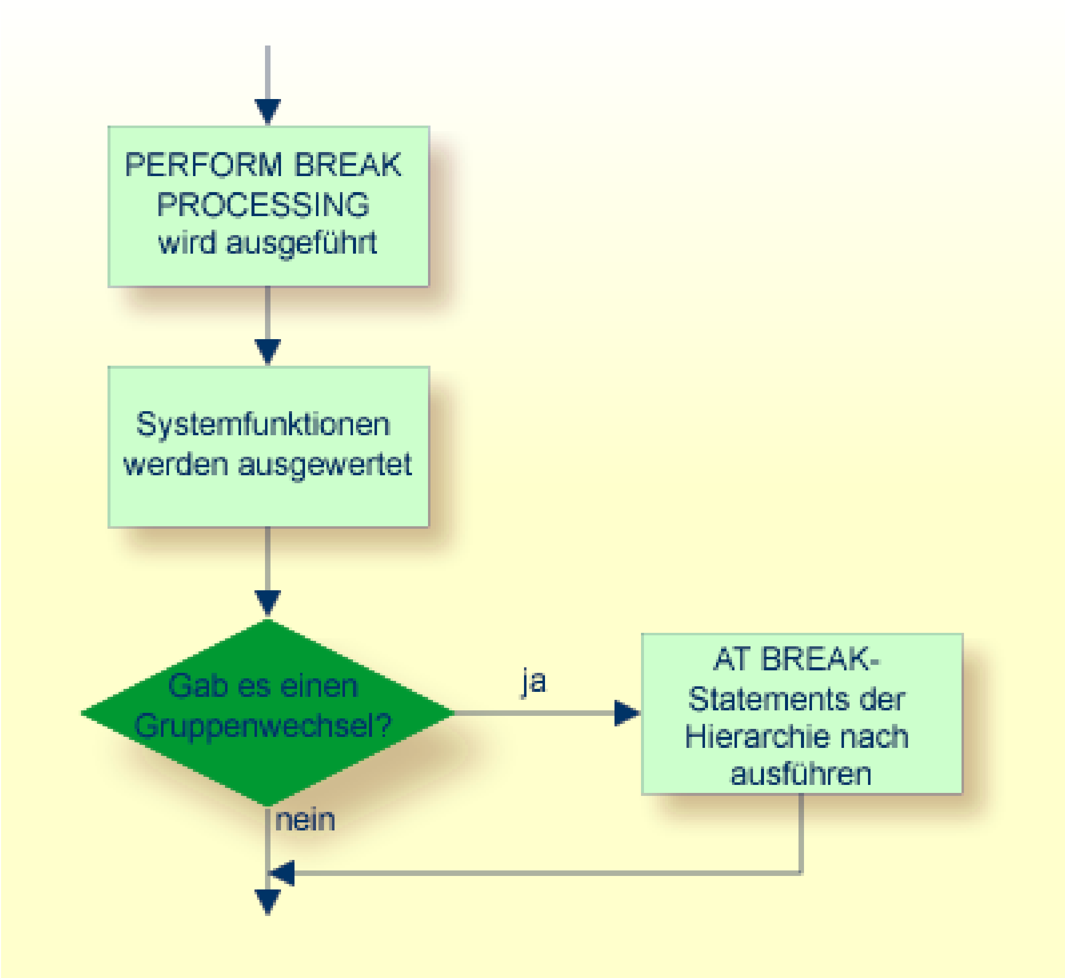
Unmittelbar nach dem PERFORM BREAK PROCESSING-Statement geben Sie einen oder mehrere AT BREAK-Statement-Blöcke an:

```
...
PERFORM BREAK PROCESSING
  AT BREAK OF field1
    statements
  END-BREAK
  AT BREAK OF field2
    statements
  END-BREAK
...
```

Wenn ein PERFORM BREAK PROCESSING-Statement ausgeführt wird, prüft Natural, ob ein Gruppenwechsel stattgefunden hat, d.h. ob der Wert des angegebenen Kontrollfeldes sich geändert hat; ist dies der Fall, dann werden die angegebenen Statements ausgeführt.

Bei PERFORM BREAK PROCESSING werden Systemfunktionen ausgewertet, *bevor* Natural prüft, ob ein Gruppenwechsel stattgefunden hat.

Die folgende Abbildung zeigt den logischen Ablauf einer programmabhängigen Gruppenwechsel-Verarbeitung:



Beispiel für PERFORM BREAK PROCESSING-Statement

```
** Example 'PERFBX01': PERFORM BREAK PROCESSING (with BREAK option
**                     in IF statement)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 SALARY (1:1)
*
1 #CNTL    (N2)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY DEPT
```

```
  AT BREAK OF DEPT              /* <- automatic break processing
    SKIP 1
    WRITE 'SUMMARY FOR ALL SALARIES         '
          'SUM:'   SUM(SALARY(1))
          'TOTAL:' TOTAL(SALARY(1))
    ADD 1 TO #CNTL
  END-BREAK
  /*
  IF SALARY (1) GREATER THAN 100000 OR BREAK #CNTL
    PERFORM BREAK PROCESSING    /* <- user-initiated break processing
    AT BREAK OF #CNTL
      WRITE 'SUMMARY FOR SALARY GREATER 100000'
            'SUM:'   SUM(SALARY(1))
            'TOTAL:' TOTAL(SALARY(1))
    END-BREAK
  END-IF
  /*
  IF SALARY (1) GREATER THAN 150000 OR BREAK #CNTL
    PERFORM BREAK PROCESSING    /* <- user-initiated break processing
    AT BREAK OF #CNTL
      WRITE 'SUMMARY FOR SALARY GREATER 150000'
            'SUM:'   SUM(SALARY(1))
            'TOTAL:' TOTAL(SALARY(1))
    END-BREAK
  END-IF
  DISPLAY NAME DEPT SALARY(1)
END-READ
END
```

Ausgabe des Programms `PERFBX01`:

```
Page      1                                                    04-12-14  14:13:35

        NAME         DEPARTMENT   ANNUAL
                        CODE      SALARY
-------------------- ---------- ----------

JENSEN               ADMA01         180000
PETERSEN             ADMA01         105000
MORTENSEN            ADMA01         320000
MADSEN               ADMA01         149000
BUHL                 ADMA01         642000

SUMMARY FOR ALL SALARIES          SUM:    1396000 TOTAL:    1396000
SUMMARY FOR SALARY GREATER 100000 SUM:    1396000 TOTAL:    1396000
SUMMARY FOR SALARY GREATER 150000 SUM:    1142000 TOTAL:    1142000
HERMANSEN            ADMA02         391500
PLOUG                ADMA02         162900

SUMMARY FOR ALL SALARIES          SUM:     554400 TOTAL:    1950400
```

```
SUMMARY FOR SALARY GREATER 100000 SUM:     554400 TOTAL:    1950400
SUMMARY FOR SALARY GREATER 150000 SUM:     554400 TOTAL:    1696400
```

49 Datenberechnungen

- COMPUTE-Statement 438
- Statements MOVE und COMPUTE 439
- Statements ADD, SUBTRACT, MULTIPLY und DIVIDE 440
- Beispiel für MOVE-, SUBTRACT- und COMPUTE-Statements 440
- COMPRESS-Statement 441
- Beispiel für die Statements COMPRESS und MOVE 442
- Beispiel für COMPRESS-Statement 443
- Mathematische Funktionen 444
- Weitere Beispiele für COMPUTE-, MOVE- und COMPRESS-Statements 445

Dieses Kapitel behandelt die zur Berechnung von Daten verwendeten arithmetischen Statements:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

Außerdem werden die folgenden Statements behandelt, die die zur Übertragung des Wertes eines Operanden in eines oder mehrere Felder benutzt werden:

- MOVE
- COMPRESS

Wichtig: Um die Verarbeitung zu optimieren, sollten **Benutzervariablen**, die in arithmetischen Statements verwendet werden, mit Format P (gepackt numerisch) definiert werden.

COMPUTE-Statement

Mit dem COMPUTE-Statement können Sie Rechenoperationen ausführen. Die folgenden Operatoren stehen Ihnen hierbei zur Verfügung:

**	Potenzierung
*	Multiplikation
/	Division
+	Addition
-	Subtraktion
( )	Logische Gruppen können mittels Klammerung gebildet werden.

Beispiel 1:

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

Der Wert des Feldes LEAVE-DUE wird mit 1,1 multipliziert und das Ergebnis in LEAVE-DUE gestellt.

Beispiel 2:

```
COMPUTE #A = SQRT (#B)
```

Die Quadratwurzel des Feldwertes von #B wird errechnet und dem Feld #A zugewiesen. SQRT (für Square Root = Quadratwurzel) ist eine von mehreren in Natural eingebauten mathematischen Funktionen, die mit den folgenden Statements verwendet werden können:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

Einen Überblick über die in Natural eingebauten mathematische Funktionen finden Sie im Abschnitt **Mathematische Funktionen**.

Beispiel 3:

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

Der erste Bonus des laufenden Jahres und das derzeitige Gehalt werden addiert, und das Ergebnis wird in das Feld #INCOME gestellt.

Statements MOVE und COMPUTE

Mit den Statements MOVE und COMPUTE stellen Sie den Wert eines Operanden in ein oder mehrere Felder. Der Operand kann eine Text/Zahlenkonstante, Benutzervariable, ein Datenbankfeld, eine Systemvariable und in bestimmten Fällen auch eine Systemfunktion sein.

Die Statements MOVE und COMPUTE unterscheiden sich in ihrer Syntax dahingehend voneinander, dass beim MOVE-Statement der zu verschiebende Wert links angegeben wird, und beim COMPUTE-Statement der zuzuweisende Wert rechts angegeben wird, wie folgende Beispiele zeigen:

Beispiele:

```
MOVE NAME TO #LAST-NAME
COMPUTE #LAST-NAME = NAME
```

Statements ADD, SUBTRACT, MULTIPLY und DIVIDE

Mit den Statements `ADD`, `SUBTRACT`, `MULTIPLY` und `DIVIDE` können Sie Rechenoperationen ausführen.

Beispiele:

```
ADD +5 -2 -1 GIVING #A
SUBTRACT 6 FROM 11 GIVING #B
MULTIPLY 3 BY 4 GIVING #C
DIVIDE 3 INTO #D GIVING #E
```

Alle 4 Statements haben eine `ROUNDED`-Option, mit der Sie gerundete Werte erhalten können.

Informationen zu Rundungsregeln entnehmen Sie dem Abschnitt ***Regeln für arithmetische Operationen***. Weitere Informationen zu diesen Statements finden Sie in der *Statements*-Dokumentation.

Beispiel für MOVE-, SUBTRACT- und COMPUTE-Statements

Das folgende Programm veranschaulicht die Verwendung von **Benutzervariablen** in arithmetischen Statements. Es berechnet Alter und Einkommen von drei Mitarbeitern und gibt die Ergebnisse aus.

```
** Example 'COMPUX01': COMPUTE
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY         (1:1)
  2 BONUS          (1:1,1:1)
*
1 #DATE            (N8)
1 REDEFINE #DATE
  2 #YEAR          (N4)
  2 #MONTH         (N2)
```

```
  2 #DAY           (N2)
1 #BIRTH-YEAR      (A4)
1 REDEFINE #BIRTH-YEAR
  2 #BIRTH-YEAR-N  (N4)
1 #AGE             (N3)
1 #INCOME          (P9)
END-DEFINE
*
MOVE *DATN TO #DATE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE EDITED BIRTH (EM=YYYY) TO #BIRTH-YEAR
  SUBTRACT #BIRTH-YEAR-N FROM #YEAR GIVING #AGE
  /*
  COMPUTE #INCOME = BONUS (1:1,1:1) + SALARY (1:1)
  /*
  DISPLAY NAME 'POSITION' JOB-TITLE #AGE #INCOME
END-READ
END
```

Ausgabe des Programms `COMPUX01`:

```
Page      1                                              04-11-11  14:15:54

        NAME                 POSITION          #AGE  #INCOME
-------------------- ------------------------- ---- ----------

JONES                MANAGER                     63      55000
JONES                DIRECTOR                    58      50000
JONES                PROGRAMMER                  48      31000
```

COMPRESS-Statement

Mit dem `COMPRESS`-Statement fassen Sie den Inhalt zweier oder mehrerer Operanden in einem einzigen alphanumerischen Feld zusammen.

Führende Nullen in einem numerischen Feld bzw. nachfolgende Leerstellen in einem alphanumerischen Feld werden unterdrückt, bevor der Feldwert in das Zielfeld übertragen wird.

Standardmäßig werden die übertragenen Werte im Zielfeld jeweils durch ein Leerzeichen voneinander getrennt. Andere Trennmöglichkeiten sind in der *Statements*-Dokumentation unter der `COMPRESS`-Statement-Option `LEAVING NO SPACE` beschrieben.

Beispiel:

```
COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
```

In diesem Beispiel werden eine Textkonstante (`'NAME:'`), ein Datenbankfeld (`FIRST-NAME`) und eine Benutzervariable (`#LAST-NAME`) mittels eines `COMPRESS`-Statements in einer Benutzervariablen (`#FULLNAME`) zusammengefasst.

Weitere Informationen zum `COMPRESS`-Statement finden Sie in der *Statements*-Dokumentation.

Beispiel für die Statements COMPRESS und MOVE

Das folgende Beispiel veranschaulicht die Benutzung der Statements `MOVE` und `COMPRESS`.

```
** Example 'COMPRX01': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
*
1 #LAST-NAME (A15)
1 #FULL-NAME (A30)
END-DEFINE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE NAME TO #LAST-NAME
  /*
  COMPRESS 'NAME:' FIRST-NAME MIDDLE-I #LAST-NAME INTO #FULL-NAME
  /*
  DISPLAY #FULL-NAME (UC==) FIRST-NAME 'I' MIDDLE-I (AL=1) NAME
END-READ
END
```

Ausgabe des Programms `COMPRX01`:

Beachten Sie vor allem die Ausgabe der mittels `COMPRESS`-Statement zusammengefassten Felder.

```
Page      1                                            04-11-11  14:15:54

         #FULL-NAME                 FIRST-NAME      I         NAME
============================== -------------------- - --------------------

NAME: VIRGINIA J JONES         VIRGINIA             J JONES
NAME: MARSHA JONES             MARSHA                 JONES
NAME: ROBERT B JONES           ROBERT               B JONES
```

Bei mehrzeiligen Ausgaben kann es sinnvoll sein, mit einem `COMPRESS`-Statement mehrere Felder/Text in einer **Benutzervariablen** zusammenzufassen.

Beispiel für COMPRESS-Statement

Im folgenden Programm werden drei **Benutzervariablen** benutzt: `#FULL-SALARY`, `#FULL-NAME` und `#FULL-CITY`. In `#FULL-SALARY` beispielsweise sind der Text `'SALARY:'` sowie die Datenbankfelder `SALARY` und `CURR-CODE` zusammengefasst. Das `WRITE`-Statement referenziert lediglich die komprimierten Variablen.

```
** Example 'COMPRX02': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY       (1:1)
  2 CURR-CODE    (1:1)
  2 CITY
  2 ADDRESS-LINE (1:1)
  2 ZIP
*
1 #FULL-SALARY   (A25)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULL-SALARY
  COMPRESS FIRST-NAME NAME                  INTO #FULL-NAME
  COMPRESS ZIP CITY                         INTO #FULL-CITY
  /*
  DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X)
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
        2/5  ADDRESS-LINE (1)
        3/5  #FULL-CITY
```

```
  SKIP 1
END-READ
END
```

Ausgabe des Programms `COMPRX02`:

```
Page      1                                                   04-11-11  14:15:54

   NAME AND ADDRESS
-----------------------

R U B I N
    SYLVIA RUBIN                   SALARY: USD 17000
    2003 SARAZEN PLACE
    10036 NEW YORK

W A L L A C E
    MARY WALLACE                   SALARY: USD 38000
    12248 LAUREL GLADE C
    10036 NEW YORK

K E L L O G G
    HENRIETTA KELLOGG              SALARY: USD 52000
    1001 JEFF RYAN DR.
    19711 NEWARK
```

Mathematische Funktionen

Bei der Verarbeitung arithmetischer Statements (`ADD`, `COMPUTE`, `DIVIDE`, `SUBTRACT`, `MULTIPLY`) unterstützt Natural die folgenden mathematischen Funktionen:

Mathematische Funktion	Natural-Systemfunktion
Absoluter Wert eines Feldes.	ABS(*field*)
Arcustangens eines Feldes.	ATN(*field*)
Kosinus eines Feldes.	COS(*field*)
Potenz eines Feldes (Exponential).	EXP(*field*)
Bruchteil (hinter dem Komma) eines Feldes.	FRAC(*field*)
Ganzzahliger Teil eines Feldes.	INT(*field*)
Natürlicher Logarithmus eines Feldes.	LOG(*field*)
Vorzeichen eines Feldes.	SGN(*field*)
Sinus eines Feldes.	SIN(*field*)
Quadratwurzel eines Feldes (Square Root).	SQRT(*field*)

Mathematische Funktion	Natural-Systemfunktion
Tangens eines Feldes.	TAN(*field*)
Numerischer Wert eines alphanumerischen Feldes.	VAL(*field*)

Weitere Einzelheiten zu mathematischen Funktionen finden Sie in der *Systemfunktionen*-Dokumentation.

Weitere Beispiele für COMPUTE-, MOVE- und COMPRESS-Statements

Siehe folgende Beispielprogramme:

- *WRITEX11 – WRITE-Statement (mit nX, n/n und COMPRESS)*
- *IFX03 - IF-Statement*
- *COMPRX03 - COMPRESS-Statement (mit Parametern LC and TC)*

50 Systemvariablen und Systemfunktionen

- Systemvariablen 448
- Systemfunktionen 450
- Beispiel für Systemvariablen und Systemfunktionen 450
- Weitere Beispiele für Systemvariablen 452
- Weitere Beispiele für Systemfunktionen 452

Dieses Kapitel beschreibt den Sinn und Zweck von Natural-Systemvariablen und Natural-Systemfunktionen, und wie sie in Natural-Programmen benutzt werden.

Systemvariablen

Folgende Themen werden behandelt:

- Sinn und Zweck von Systemvariablen
- Charakteristika von Systemvariablen
- Aufteilung von Systemvariablen nach Funktionen

Sinn und Zweck von Systemvariablen

Systemvariablen werden zur Anzeige von System-Informationen benutzt. Sie können an einem beliebigen Punkt in einem Natural-Programm referenziert werden.

Natural-Systemvariablen liefern Variablen-Informationen, z.B. zur aktuellen Natural-Session:

- die aktuelle Library
- die Benutzer-ID und die Terminal-ID
- den aktuellen Status eines Schleifendurchlaufs
- den aktuellen Verarbeitungs-Status von Reports
- das aktuelle Datum und die aktuelle Uhrzeit

Die typische Benutzung von Systemvariablen wird im ***Beispiel für Systemvariablen und Systemfunktionen*** weiter unten und in den in der Library `SYSEXPG` enthaltenen Beispielen veranschaulicht.

Die in einer Systemvariable enthaltenen Informationen können in Natural-Programmen unter Angabe der betreffenden Systemvariablen benutzt werden. Beispielsweise können Datums- und Zeit-Systemvariablen in einem `DISPLAY`-, `WRITE`-, `PRINT`-, `MOVE`- oder `COMPUTE`-Statement angegeben werden.

Charakteristika von Systemvariablen

Die Namen der Systemvariablen beginnen mit einem Stern (*).

Format/Länge

Folgende Abkürzungen werden verwendet:

Format	
A	Alphanumerisch
B	Binär
D	Datum
I	Integer (Ganzzahl)
L	Logisch
N	Numerisch (ungepackt)
P	Numerisch (gepackt)
T	Zeit

Inhalt änderbar

In den einzelnen Beschreibungen verweist dies darauf, ob Sie in einem Natural-Programm der Systemvariablen einen anderen Wert zuweisen können, d.h. ihren von Natural generierten Inhalt überschreiben können.

Aufteilung von Systemvariablen nach Funktionen

Die Natural-Systemvariablen sind wie folgt unterteilt:

- Anwendungsbezogene Systemvariablen
- Datums- und Zeit-Systemvariablen
- Eingabe/Ausgabe-bezogene Systemvariablen
- Natural-Umgebungsbezogene Systemvariablen
- System-Umgebungsbezogene Systemvariablen
- XML-bezogene Systemvariablen

Ausführliche Beschreibungen aller Systemvariablen finden Sie in der *Systemvariablen* -Dokumentation.

Systemfunktionen

Natural-Systemfunktionen sind in Natural eingebaute Funktionen, mit denen Sie statistische und mathematische Informationen über die gelesenen Daten erhalten können. Sie können eingesetzt werden, nachdem ein Datensatz gelesen worden ist, aber vor einem Gruppenwechsel.

Systemfunktionen können in WRITE-, DISPLAY-, PRINT-, COMPUTE- oder MOVE-Statements in Verbindung mit AT END OF PAGE-, AT END OF DATA- und AT BREAK-Statements benutzt werden.

Im Falle eines AT END OF PAGE-Statements muss das jeweilige DISPLAY-Statement eine GIVE SYSTEM FUNCTIONS-Klausel enthalten (wie im **Beispiel** gezeigt).

Es gibt folgende nach Funktionen aufgeteilten Systemfunktionen:

- Systemfunktionen zur Verwendung in Verarbeitungsschleifen
- Mathematische Funktionen
- Verschiedene Funktionen

Weitere Informationen zu Systemfunktionen finden Sie in der *Systemfunktionen*-Dokumentation.

Siehe dort auch *Natural-Systemfunktionen für Verarbeitungsschleifen* in der *Systemfunktionen*-Dokumentation.

Die typische Benutzung von Systemfunktionen ist in den folgenden Beispielprogrammen und in den in der Library SYSEXPG enthaltenen Beispielen erläutert.

Beispiel für Systemvariablen und Systemfunktionen

Das folgende Beispielprogramm veranschaulicht die Verwendung von Systemvariablen und Systemfunktionen:

```
** Example 'SYSVAX01': System variables and system functions
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME    (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS   (1:1)
```

```
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED 'EMPLOYEE SALARY REPORT AS OF' *DAT4E /
*
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1:1)
  AT START OF DATA
    WRITE 'REPORT CREATED AT:' *TIME 'HOURS' /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
*
AT END OF PAGE
  WRITE 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END
```

Erläuterung:

- Die Systemvariable `*DATE` wird mit dem `WRITE TITLE`-Statement ausgegeben.
- Die Systemvariable `*TIME` wird mit dem `AT START OF DATA`-Statement ausgegeben.
- Die Systemfunktion `OLD` wird im `AT END OF DATA`-Statement benutzt.
- Die Systemfunktion `AVER` wird im `AT END OF PAGE`-Statement benutzt.

Ausgabe des Programms `SYSVAX01`:

Beachten Sie, wie die Systemvariablen und Systemfunktionen angezeigt werden:

```
EMPLOYEE SALARY REPORT AS OF 11/11/2004

     NAME           CURRENT                INCOME
                   POSITION
                                CURRENCY   ANNUAL     BONUS
                                  CODE     SALARY
--------------- --------------- -------- ---------- ----------

REPORT CREATED AT: 14:15:55.0 HOURS

DUYVERMAN       PROGRAMMER      USD           34000          0
PRATT           SALES PERSON    USD           38000       9000
MARKUSH         TRAINEE         USD           22000          0

LAST PERSON SELECTED: MARKUSH

AVERAGE SALARY:      31333
```

Weitere Beispiele für Systemvariablen

Siehe folgende Beispielprogramme:

- *EDITMX05 - Editiermaske (EM-Angaben für Datums- und Uhrzeit-Systemvariablen)*
- *READX04 – READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)*
- *WTITLX01 – WRITE TITLE-Statement (mit *PAGE–NUMBER)*

Weitere Beispiele für Systemfunktionen

Siehe folgende Beispielprogramme:

- *ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)*
- *ATENPX01 - AT END OF PAGE-Statement (mit der durch GIVE SYSTEM FUNCTIONS in DISPLAY verfügbaren Systemfunktion)*

51 Natural-Stack

- Verwendung des Natural-Stack 454
- Stack-Verarbeitung 454
- Daten im Stack ablegen 455
- Stack-Inhalt löschen 456

Der Natural-Stack ist eine Art „Zwischenablage", in der Sie Natural-Kommandos, Benutzerkommandos und Daten für ein INPUT-Statement speichern können.

Verwendung des Natural-Stack

So können Sie häufig nacheinander ausgeführte Funktionen, wie beispielsweise eine Abfolge von Logon-Kommandos, die häufig in der gleichen Reihenfolge ausgeführt werden, speichern.

Der Stack ist mit einem Stapel vergleichbar: die Daten/Kommandos werden aufeinander „gestapelt" und können sowohl oben auf dem Stack als auch unten im Stack abgelegt werden. Die gespeicherten Daten/Kommandos können nur in der gestapelten Reihenfolge verarbeitet werden, und zwar von oben nach unten.

Mit der Systemvariable *DATA können Sie sich in einem Programm den Inhalt des Stack anzeigen lassen (weitere Informationen siehe *Systemvariablen*-Dokumentation).

Stack-Verarbeitung

Die Verarbeitung der im Stack gespeicherten Kommandos und Daten ist abhängig von der jeweils ausgeführten Funktion.

Falls ein Kommando einzugeben ist, d.h. falls als nächstes die NEXT-Zeile erscheinen müsste, sucht Natural den Stack von oben nach unten nach einem Kommando ab; wird ein Kommando gefunden, so wird die NEXT-Zeile unterdrückt, das Kommando gelesen und aus dem Stack gelöscht. Das Kommando wird ausgeführt, als wäre es von Hand in der NEXT-Zeile eingegeben worden.

Falls ein INPUT-Statement ausgeführt wird, das Eingabefelder enthält, sucht Natural, bevor der INPUT-Schirm angezeigt wird, den Stack nach Eingabedaten ab und übergibt diese automatisch an das INPUT-Statement (und zwar unter Delimiter-Mode-Logik). Natural überprüft, ob es sich um für das betreffende INPUT-Statement gültige Eingabedaten handelt; anschließend löscht es die Daten aus dem Stack. Siehe auch *INPUT-Daten aus dem Natural-Stack* in der Beschreibung des INPUT-Statements.

Falls ein INPUT-Statement mit Stack-Daten ausgeführt wird und dieses INPUT-Statement durch ein REINPUT-Statement nochmals ausgeführt wird, wird der INPUT-Schirm angezeigt, und zwar mit den gleichen Stack-Daten wie beim ersten Mal. Bei einem REINPUT-Statement werden keine weiteren Daten vom Stack gelesen.

Wird ein Natural-Programm normal beendet, werden die zuoberst gelagerten Daten im Stack soweit gelöscht, bis sich entweder oben im Stack wieder ein Kommando befindet oder der Stack ganz geleert ist. Wird ein Natural-Programm aufgrund eines Fehlers oder mit dem Terminalkommando %% abgebrochen, wird der gesamte Inhalt des Stacks gelöscht.

Daten im Stack ablegen

Es gibt folgende Möglichkeiten, Daten bzw. Kommandos im Stack abzulegen:

- STACK-Parameter
- STACK-Statement
- FETCH- und RUN-Statements

STACK-Parameter

Sie können den Natural-Profilparameter STACK benutzen, um Daten oder Kommandos im Stack abzulegen. Der STACK-Parameter, der in der Natural *Parameter-Referenz*-Dokumentation beschrieben ist, kann bei der Installation von Natural vom Natural-Administrator im Natural-Parametermodul gesetzt werden. Sie können den STACK-Parameter auch als dynamischen Parameter beim Aufruf von Natural angeben.

Werden Daten/Kommandos mit dem STACK-Parameter im Stack abgelegt, so müssen mehrere Kommandos mit einem Semikolon (;) voneinander getrennt werden. Einem Kommando, das innerhalb einer Reihe von Daten- bzw. Kommandoelementen übergeben wird, muss ein Semikolon vorangestellt werden.

Daten für mehrere INPUT-Statements müssen mit einem Doppelpunkt (:) voneinander getrennt werden. Einer Datenkette, die von einem weiteren INPUT-Statement gelesen werden soll, muss jeweils ein Doppelpunkt vorangestellt werden. Soll ein Kommando, das Parameter erfordert, im Stack abgelegt werden, werden das Kommando und die dazugehörigen Parameter nicht durch einen Doppelpunkt voneinander getrennt.

Doppelpunkt und Semikolon dürfen nicht in den für das INPUT-Statement bestimmten Daten selbst auftauchen, da sie als Trennzeichen interpretiert werden.

STACK-Statement

Innerhalb eines Natural-Programms können Sie das STACK-Statement verwenden, um Daten oder Kommandos im Stack abzulegen. Die in einem STACK-Statement angegebenen Datenelemente können nur für ein einziges INPUT-Statement verwendet werden; d.h. Sie müssen mehrere STACK-Statements verwenden, wenn Sie Daten für mehrere INPUT-Statements im Stack ablegen wollen.

Daten können entweder formatiert oder unformatiert im Stack abgelegt werden:

- Werden unformatierte Daten aus dem Stack gelesen, werden sie im Delimiter-Modus interpretiert, wobei die mit den Session-Parametern IA (Input Assign) und ID (Input Delimiter) festgelegten Zeichen als Input-Zuweisungszeichen bzw. -Trennzeichen verarbeitet werden.

- Formatiert im Stack gelagerte Daten werden nach Feldinhalten getrennt und Feld für Feld an die Eingabefelder des betreffenden `INPUT`-Statements übergeben. Falls die im Stack abzulegenden Daten Begrenzungs-, Steuer- oder DBCS-Zeichen enthalten, sollten sie, um eine unbeabsichtigte Interpretation dieser Zeichen zu vermeiden, formatiert im Stack abgelegt werden.

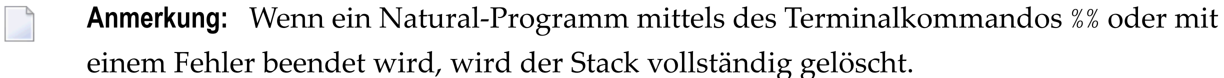
Eine ausführliche Beschreibung des Statements `STACK` finden Sie in der *Statements*-Dokumentation.

FETCH- und RUN-Statements

Werden bei der Ausführung eines `FETCH`- oder `RUN`-Statements Parameter an das aufgerufene Programm übergeben, so werden diese Parameter oben auf dem Stack abgelegt.

Stack-Inhalt löschen

Der Inhalt des Stacks kann mit dem Statement `RELEASE` gelöscht werden. Eine ausführliche Beschreibung des Statements finden Sie in der *Statements*-Dokumentation.



Anmerkung: Wenn ein Natural-Programm mittels des Terminalkommandos `%%` oder mit einem Fehler beendet wird, wird der Stack vollständig gelöscht.

52 Datumsinformationen verarbeiten

- Editiermasken für Datumsfelder and Datumssystemvariablen 458
- Standard-Editiermaske für Datum — der DTFORM-Parameter 458
- Datumsformat für alphanumerische Darstellung — der DF-Parameter 459
- Datumsformat für Ausgabe — der DFOUT-Parameter 462
- Datumsformat für Stack — der DFSTACK-Parameter 463
- Year Sliding Window — der YSLW-Parameter 464
- Kombinationen von DFSTACK und YSLW 467
- Year Fixed Window 469
- Datumsformat für Standard-Seitenüberschriften — der DFTITLE-Parameter 470

Dieses Kapitel behandelt verschiedene Aspekte der Behandlung von Datumsinformationen in Ihren Natural-Anwendungen.

Editiermasken für Datumsfelder and Datumssystemvariablen

Wenn Sie den Wert eines Datumsfeldes in einer bestimmten Form ausgeben möchten, geben Sie normalerweise für das Feld eine **Editiermaske** an. Mit der Editiermaske bestimmen Sie Zeichen für Zeichen, wie die Ausgabe aussehen soll.

Falls Sie das aktuelle Datum in einer bestimmten Form verwenden möchten, brauchen Sie hierfür kein Datumsfeld mit einer entsprechenden Editiermaske zu definieren; stattdessen können Sie einfach eine *Datumssystemvariable* verwenden. Natural bietet verschiedene Datumssystemvariablen, die alle das aktuelle Datum in unterschiedlichen Darstellungsformen enthalten. Bei manchen dieser Darstellungsformen ist die Jahreszahl-Angabe zweistellig, bei manchen vierstellig.

Weitere Informationen sowie eine Liste aller Datumssystemvariablen finden Sie in der *Systemvariablen*-Dokumentation.

Standard-Editiermaske für Datum — der DTFORM-Parameter

Der Profilparameter `DTFORM` bestimmt das Standardformat, das für Datumsangaben als Teil von Natural-Standard-Reporttiteln, für Datumskonstanten und für Datumseingaben gilt.

Dieses Datumsformat bestimmt die Reihenfolge der Angaben für Tag, Monat und Jahr sowie die Trennzeichen, die zwischen diesen Angaben stehen müssen.

Mögliche `DTFORM`-Einstellungen sind:

Einstellung	Datumsformat*	Beispiel
DTFORM=I	*yyyy-mm-dd*	2005-12-31
DTFORM=G	*dd.mm.yyyy*	31.12.2005
DTFORM=E	*dd/mm/yyyy*	31/12/2005
DTFORM=U	*mm/dd/yyyy*	12/31/2005

* *dd* = day/Tag, *mm* = month/Monat, *yyyy* = year/Jahr.

Der `DTFORM`-Parameter kann in der Natural-Parameterdatei oder dynamisch beim Aufruf von Natural gesetzt werden. Standardmäßig gilt `DTFORM=I`.

Datumsformat für alphanumerische Darstellung — der DF-Parameter

Wenn eine Editiermaske angegeben ist, wird die Darstellung des Feldwertes durch die Editiermaske bestimmt. Wenn keine Editiermaske angegeben ist, wird die Darstellung des Feldwertes durch den Session-Parameter DF in Kombination mit dem DTFORM-Profilparameter bestimmt.

Mit dem DF-Parameter können Sie eine der folgenden Datumsdarstellungen wählen:

DF=S	8-Byte-Darstellung mit 2-stelliger Jahreskomponente und Begrenzungszeichen (*yy-mm-dd*).
DF=I	8-Byte-Darstellung mit 4-stelliger Jahreskomponente ohne Begrenzungszeichen (*yyyymmdd*).
DF=L	10-Byte-Darstellung mit 4-stelliger Jahreskomponente und Begrenzungszeichen (*yyyy-mm-dd*).

Bei jeder Darstellung wird die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die zu verwendenden Begrenzungszeichen durch den DTFORM-Parameter bestimmt.

Standardmäßig gilt DF=S (außer bei INPUT-Statements; siehe unten).

Der DF-Parameter wird bei der Kompilierung ausgewertet.

Der DF-Parameter gilt bei folgenden Statements und in folgenden Situationen:

- FORMAT
- INPUT, DISPLAY, WRITE und PRINT auf Statement- und Elementebene (Feldebene)
- MOVE, COMPRESS, STACK, RUN und FETCH auf Elementebene (Feldebene)

Bei Angabe in einem dieser Statements hat der DF-Parameter folgende Auswirkung:

Statement	Auswirkung des DF-Parameters
DISPLAY, WRITE, PRINT	Wenn der Wert einer Datumsvariablen mit einem dieser Statements ausgegeben wird, wird der Wert in alphanumerische Darstellung umgesetzt, bevor er ausgegeben wird. Der DF-Parameter bestimmt, welche Darstellung dafür verwendet wird.
MOVE, COMPRESS	Wenn der Wert einer Datumsvariablen mit einem MOVE- oder COMPRESS-Statement in ein alphanumerisches Feld übertragen wird, wird der Wert in alphanumerische Darstellung umgesetzt, bevor er übertragen wird. Der DF-Parameter bestimmt, welche Darstellung dafür verwendet wird.

Statement	Auswirkung des DF-Parameters
STACK, RUN, FETCH	Wenn der Wert einer Datumsvariablen auf dem Stack abgelegt wird, wird er in alphanumerische Darstellung umgesetzt, bevor er auf dem Stack abgelegt wird. Der DF-Parameter bestimmt, welche Darstellung dafür verwendet wird. Dasselbe gilt, wenn eine Datumsvariable als Parameter in einem FETCH- oder RUN-Statement angegeben ist (da diese Parameter ebenfalls über den Stack übergeben werden).
INPUT	Wenn eine Datumsvariable in einem INPUT-Statement verwendet wird, bestimmt der DF-Parameter, in welcher Form ein Wert in dieses Feld eingegeben werden muss. Wenn dagegen eine Datumsvariable, für die kein DF-Parameter angegeben ist, in einem INPUT-Statement verwendet wird, kann das Datum entweder mit 2-stelliger Jahresangabe und Begrenzungszeichen oder mit 4-stelliger Jahresangabe ohne Begrenzungszeichen eingegeben werden. Auch in diesem Fall werden die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die zu verwendenden Begrenzungszeichen durch den DTFORM-Parameter bestimmt.

Anmerkung: Bei DF=S stehen nur 2 Stellen für die Jahresangabe zur Verfügung; d.h. selbst wenn der Datumswert das Jahrhundert enthielte, gingen diese Informationen bei der Umsetzung verloren. Um die Jahrhundert-Angabe zu behalten, setzen Sie DF=I oder DF=L.

Beispiele für DF-Parameter beim WRITE-Statements

Diese Beispiele gehen von DTFORM=G aus.

```
/* DF=S (default)
WRITE *DATX   /* Output has this format: dd.mm.yy
END
```

```
FORMAT DF=I
WRITE *DATX   /* Output has this format: ddmmyyyy
END
```

```
FORMAT DF=L
WRITE *DATX   /* Output has this format: dd.mm.yyyy
END
```

Beispiel für DF-Parameter beim MOVE-Statement

Dieses Beispiel geht von `DTFORM=E` aus.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'31/12/2005'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE        TO #ALPHA /* Result: #ALPHA contains 31/12/05
MOVE #DATE (DF=I) TO #ALPHA /* Result: #ALPHA contains 31122005
MOVE #DATE (DF=L) TO #ALPHA /* Result: #ALPHA contains 31/12/2005
...
```

Beispiel für DF-Parameter beim STACK-Statement

Dieses Beispiel geht von `DTFORM=I` aus.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2005-12-31'>
  1 #ALPHA1(A10)
  1 #ALPHA2(A10)
  1 #ALPHA3(A10)
END-DEFINE
...
STACK TOP DATA #DATE (DF=S) #DATE (DF=I) #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2 #ALPHA3
...
/* Result: #ALPHA1 contains 05-12-31
/*         #ALPHA2 contains 20051231
/*         #ALPHA3 contains 2005-12-31
...
```

Beispiel für DF-Parameter beim INPUT-Statement

Dieses Beispiel geht von DTFORM=I aus.

```
DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Input must have this format: yy-mm-dd
      #DATE2 (DF=I) /* Input must have this format: yyyymmdd
      #DATE3 (DF=L) /* Input must have this format: yyyy-mm-dd
      #DATE4        /* Input must have this format: yy-mm-dd or yyyymmdd
...
```

Datumsformat für Ausgabe — der DFOUT-Parameter

Der Session- bzw. Profilparameter DFOUT gilt nur für Datumsfelder in INPUT-, DISPLAY-, PRINT- und WRITE-Statements, für die keine Editiermaske angegeben ist und für die kein DF-Parameter gilt.

Bei Datumsfeldern, die mit einem INPUT-, DISPLAY-, PRINT- oder WRITE-Statement ausgegeben werden und für die weder eine Editiermaske angegeben noch ein DF-Parameter gesetzt ist, bestimmt der Profil/Session-Parameter DFOUT die Form, in der die Feldwerte angezeigt werden.

Mögliche DFOUT-Einstellungen sind:

DFOUT=S	Datumsvariablen werden mit 2-stelliger Jahreskomponente und mit durch den DTFORM-Parameter bestimmten Begrenzungszeichen angezeigt (*yy-mm-dd*).
DFOUT=I	Datumsvariablen werden mit mit 4-stelliger Jahreskomponente und ohne Begrenzungszeichen angezeigt (*yyyymmdd*).

Standardmäßig gilt DFOUT=S. Bei beiden DFOUT-Einstellungen wird die Reihenfolge der Tages-, Monats- und Jahreskomponenten in den Datumswerten durch den DTFORM-Parameter bestimmt.

Die Länge eines Datumsfeldes wird durch die DFOUT-Einstellung nicht beeinflusst, da jede der beiden Datumswert-Darstellungen in ein 8 Byte langes Feld passt.

Der DFOUT-Parameter kann in der Natural-Parameterdatei dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

Dieses Beispiel geht von DTFORM=I aus.

```
DEFINE DATA LOCAL
1 #DATE  (D) INIT <D'2005-12-31'>
END-DEFINE
...
WRITE #DATE        /* Output if DFOUT=S is set ...: 05-12-31
                   /* Output if DFOUT=I is set ...: 20051231
WRITE #DATE (DF=L) /* Output (regardless of DFOUT): 2005-12-31
...
```

Datumsformat für Stack — der DFSTACK-Parameter

Der Session- bzw. Profilparameter DFSTACK gilt nur für Datumsfelder, die in STACK-, FETCH- und RUN-Statements verwendet werden und für die kein DF-Parameter angegeben ist.

Der DFSTACK-Parameter bestimmt die Form, in der die Werte von Datumsvariablen mit einem STACK-, RUN- oder RUN-Statement auf dem Natural-Stack abgelegt werden.

Mögliche DFSTACK-Einstellungen sind:

DFSTACK=S	Datumsvariablen werden mit zweistelliger Jahreskomponente und mit durch den DTFORM-Parameter bestimmten Begrenzungszeichen auf dem Stack abgelegt (*yy-mm-dd*).
DFSTACK=C	Wie DFSTACK=S. Allerdings wird eine Änderung des Jahrhunderts zur Laufzeit abgefangen.
DFSTACK=I	Datumsvariablen werden mit 4-stelliger Jahreskomponente und ohne Begrenzungszeichen auf dem Stack abgelegt (*yyyymmdd*).

Standardmäßig gilt DFSTACK=S. Diese Einstellung bewirkt, dass die Jahrhundert-Informationen nicht mit abgelegt werden (und verlorengehen), wenn ein Datumswert auf dem Stack abgelegt wird.

Wenn dann der Wert vom Stack in eine andere Datumsvariable eingelesen wird, wird entweder als Jahrhundert das aktuelle Jahrhundert genommen oder das Jahrhundert durch die Einstellung des YSLW-Parameters (siehe **unten**) bestimmt. Das kann dazu führen, dass das Jahrhundert ein anderes ist als im ursprünglichen Datumswert, ohne dass Natural in diesem Fall einen Fehler ausgibt.

DFSTACK=C funktioniert insofern wie DFSTACK=S, dass ein Datumswert ohne Jahrhundert- Informationen auf dem Stack abgelegt wird. Wenn allerdings der Wert vom Stack gelesen wird und das ermittelte Jahrhundert nicht mit dem des ursprünglichen Datumswerts identisch ist (entweder

aufgrund des YSLW-Parameters oder weil das ursprüngliche Jahrhundert nicht das aktuelle ist), gibt Natural einen Laufzeitfehler aus.

Anmerkung: Dieser Laufzeitfehler wird bereits ausgegeben, sobald der Wert auf dem Stack abgelegt wird.

Mit DFSTACK=I können Sie einen Datumswert in einer Länge von 8 Bytes auf dem Stack ablegen, ohne dass die Jahrhundert-Informationen verlorengehen.

Der DFSTACK-Parameter kann im Natural-Parametermodul (bzw. der -Parameterdatei) dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

Dieses Beispiel geht von DTFORM=I und YSLW=0 aus.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2005-12-31'>
  1 #ALPHA1(A8)
  1 #ALPHA2(A10)
END-DEFINE
...
STACK TOP DATA #DATE #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2
...
/* Result if DFSTACK=S or =C is set: #ALPHA1 contains 05-12-31
/* Result if DFSTACK=I is set .....: #ALPHA1 contains 20051231
/* Result (regardless of DFSTACK) .: #ALPHA2 contains 2005-12-31
...
```

Year Sliding Window — der YSLW-Parameter

Mit dem Profilparameter YSLW können Sie das Jahrhundert eines zweistelligen Jahr-Wertes bestimmen.

Der YSLW-Parameter kann in der Natural-Parameterdatei oder dynamisch beim Aufrufen von Natural gesetzt werden. Er wird zur Laufzeit ausgewertet, wenn ein alphanumerischer Datumswert mit einer zweistelligen Jahreskomponente in eine Datumsvariable übertragen wird. Dies betrifft Datumswerte, die:

- mit der **mathematischen Funktion** VAL(*field*) verwendet werden,
- mit der Option IS(D) in einer logischen Bedingung verwendet werden,

- vom **Stack** als Eingabedaten gelesen werden,
- in ein Feld als Eingabedaten eingegeben werden.

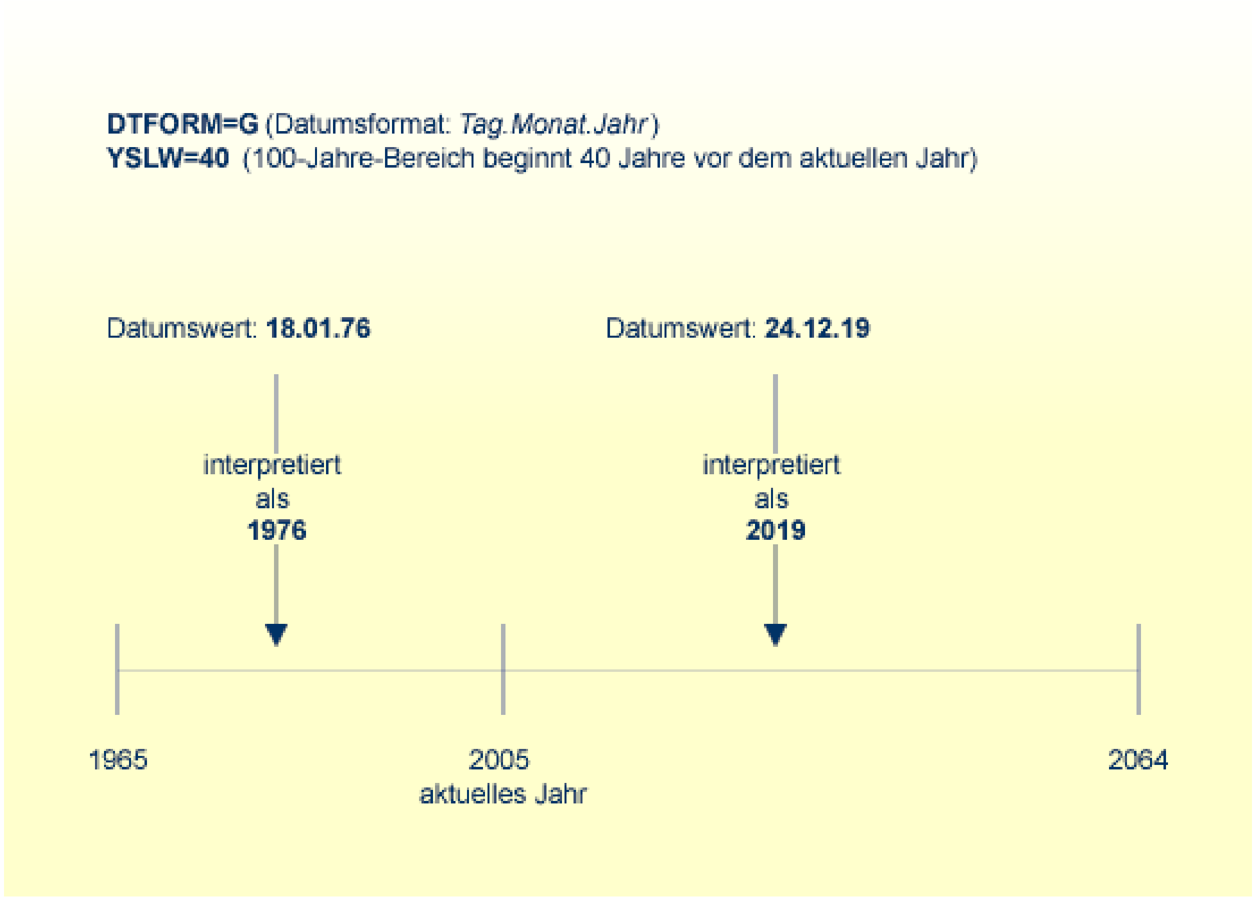
Der `YSLW`-Parameter bestimmt den Bereich von Jahren, der von einem sogenannten „Year Sliding Window“ abgedeckt wird. Der „Sliding Window-“Mechanismus geht davon aus, dass ein Datum mit einem zweistelligen Jahr innerhalb eines „Fensters“ von 100 Jahren liegt. Innerhalb dieser 100 Jahre kann jeder zweistellige Jahr-Wert eindeutig einem bestimmten Jahrhundert zugeordnet werden.

Mit dem `YSLW`-Parameter legen Sie fest, mit wievielen Jahren in der Vergangenheit der 100-Jahre-Bereich anfangen soll: das erste Jahr des Fensterbereichs ergibt sich aus dem aktuellen Jahr minus dem `YSLW`-Wert.

Mögliche Werte des `YSLW`-Parameters sind 0 bis 99. Der Standardwert ist `YSLW=0`, d.h. der „Sliding Window“-Mechanismus ist nicht aktiv; bei einem zweistelligen Jahr wird dann angenommen, dass es im aktuellen Jahrhundert liegt.

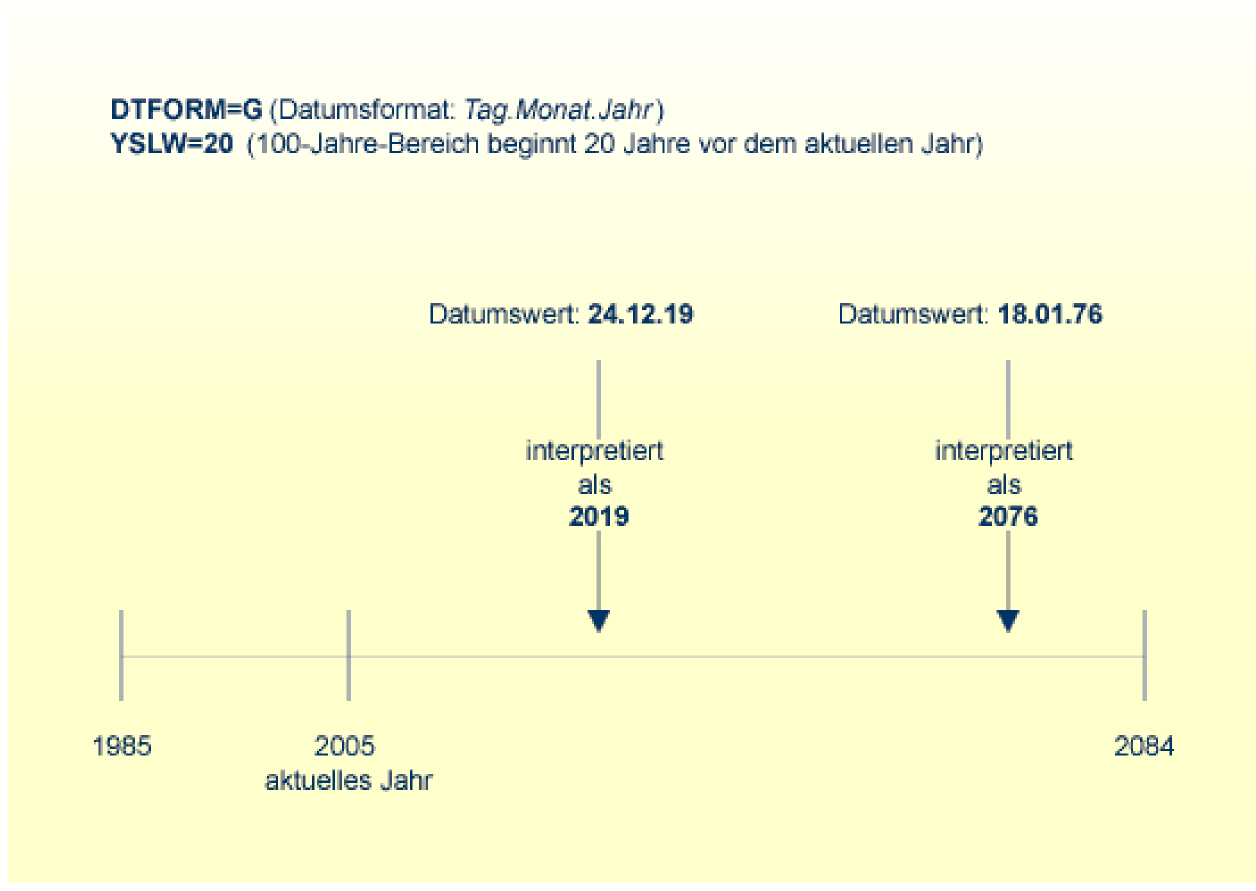
Beispiel 1:

Wenn das aktuelle Jahr `2005` ist und Sie `YSLW=40` angeben, deckt das „Sliding Window“ die Jahre `1965` bis `2064` ab. Ein zweistelliger Jahr-Wert *nn* von `65` bis `99` wird dementsprechend als `19`*nn* interpretiert, während ein zweistelliger Jahr-Wert nn von `00` bis `64` als `20`*nn* interpretiert wird.



Beispiel 2:

Wenn das aktuelle Jahr `2005` ist und Sie `YSLW=20` angeben, deckt das „Sliding Window“ die Jahre `1985` bis `2084` ab. Ein zweistelliger Jahr-Wert *nn* von `85` bis `99` wird dementsprechend als `19`*nn* interpretiert, während ein zweistelliger Jahr-Wert *nn* von `00` bis `84` als `20`*nn* interpretiert wird.



Kombinationen von DFSTACK und YSLW

Die folgenden Beispiele veranschaulichen die Wirkungen verschiedener Kombinationen der Parameter `DFSTACK` und `YSLW`.

Anmerkung: Alle Beispiele gehen von `DTFORM=I` aus.

Beispiel 1:

Dieses Beispiel geht vom aktuellen Jahr `2005` und folgenden Parametereinstellungen aus:

DFSTACK=S (Standardeinstellung) und YSLW=20

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 2056
...
/* Result: #DATE2 contains 2056-12-31
           even if #DATE1 is set to <D'2156-12-31'>
```

In diesem Fall ist das „Year Sliding Window“ unpassend gesetzt, so dass die Jahrhundert-Informationen sich (unbeabsichtigterweise) ändern.

Beispiel 2:

Dieses Beispiel geht vom aktuellen Jahr 2005 und folgenden Parametereinstellungen aus:

DFSTACK=S (Standardeinstellung) und YSLW=60

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
           even if #DATE1 is set to <D'2056-12-31'>
```

In diesem Fall ist das „Year Sliding Window“ passend gesetzt, so dass die urspünglichen Jahrhundert- Informationen korrekt wiederhergestellt werden.

Beispiel 3:

Dieses Beispiel geht vom aktuellen Jahr 2005 und folgenden Parametereinstellungen aus:

DFSTACK=C und YSLW=0 (Standardeinstellung)

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* 56 is assumed to be in current century -> 1956
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)
```

In diesem Fall ändern sich (unbeabsichtigterweise) die Jahrhundert-Informationen. Allerdings wird diese Änderung durch die Parametereinstellung DFSTACK=C abgefangen.

Beispiel 4:

Dieses Beispiel geht vom aktuellen Jahr 2005 und folgenden Parametereinstellungen aus:

DFSTACK=C und YSLW=60 (Standardeinstellung)

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)
```

In diesem Fall ändern sich die Jahrhundert-Informationen aufgrund des „Year Sliding Window“. Allerdings wird diese Änderung durch die Parametereinstellung DFSTACK=C abgefangen.

Year Fixed Window

Informationen zu diesem Thema entnehmen Sie der Beschreibung des Profilparameters YSLW.

Datumsformat für Standard-Seitenüberschriften — der DFTITLE-Parameter

Der Session- bzw. Profilparameter DFTITLE bestimmt die Form des Datums in einer Standard-**Seitenüberschrift** (wie sie mit einem DISPLAY-, WRITE- oder PRINT-Statement ausgegeben wird).

DFTITLE=S	Das Datum wird mit zweistelliger Jahreskomponente und Begrenzungszeichen ausgegeben (*yy-mm-dd*).
DFTITLE=L	Das Datum wird mit vierstelliger Jahreskomponente und Begrenzungszeichen ausgegeben (*yyyy-mm-dd*).
DFTITLE=I	Das Datum wird mit vierstelliger Jahreskomponente ohne Begrenzungszeichen ausgegeben (*yyyymmdd*).

Bei jeder dieser Ausgabeformen werden die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die verwendeten Begrenzungszeichen durch den DTFORM-Parameter bestimmt.

Der DFTITLE-Parameter kann in der Natural-Parameterdatei dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

Dieses Beispiel geht von DTFORM=I aus.

```
WRITE 'HELLO'
END
/*
/* Date in page title if DFTITLE=S is set ...: 05-10-31
/* Date in page title if DFTITLE=L is set ...: 2005-10-31
/* Date in page title if DFTITLE=I is set ...: 20051031
```

Anmerkung: Der DFTITLE-Parameter hat keine Auswirkungen auf benutzerdefinierte Seitenüberschriften, wie sie mit einem WRITE TITLE-Statement angegeben werden.

53 Text-Notation

- Mit einem Statement zu benutzenden Text definieren — die 'text'-Notation 472
- Vor einem Feldwert n-mal anzuzeigendes Zeichen definieren — die 'c'(n)-Notation 474

In den Statements `INPUT`, `DISPLAY`, `WRITE`, `WRITE TITLE` oder `WRITE TRAILER` können Sie Text-Notation benutzen, um einen in Verbindung mit einem solchen Statement zu benutzenden Text zu definieren.

Mit einem Statement zu benutzenden Text definieren — die 'text'-Notation

Der mit dem Statement zu benutzende Text (z.B. eine Aufforderungsmeldung für den Benutzer) muss entweder in Apostrophen (') oder in Anführungszeichen (") stehen.

Vorsicht: Verwechseln Sie doppelte Apostrophe ('') nicht mit einem Anführungszeichen (").

In Anführungszeichen stehender Text kann automatisch von Klein- in Großbuchstaben konvertiert werden. Um die automatische Konvertierung auszuschalten, ändern Sie die Einstellungen im Editor-Profil. Einzelheiten entnehmen Sie dem Abschnitt *Programm-Editor-Optionen*, **Ignore text constants** and **Uppercase translation** (in der Dokumentation *Natural Studio benutzen*).

Der '*text*' darf 1 bis 72 Zeichen lang sein und darf nicht über das Ende einer Sourcecode-Zeile hinausgehen.

Textelemente können mittels eines Bindestriches verkettet werden.

Beispiele:

```
DEFINE DATA LOCAL
1 #A(A10)
END-DEFINE

INPUT 'Input XYZ' (CD=BL) #A
WRITE '=' #A
WRITE 'Write1 ' - 'Write2 ' - 'Write3' (CD=RE)
END
```

Apostrophe als Teil eines Textelements benutzen

Es gilt Folgendes, wenn der Natural-Profilparameter `TQ` (Translate Quotation Marks = Anführungszeichen konvertieren) auf `ON` gesetzt ist. Dies ist die Standardstellung.

Für ein Apostroph, das Teil eines in Apostrophen stehenden *text*-Elements ist, schreiben Sie entweder doppelte Apostrophe ('') oder ein einzelnes Anführungszeichen ("); beides wird dann bei der Ausgabe in ein einzelnes Apostroph umgesetzt.

Für ein Apostroph, das Teil eines in Anführungszeichen stehenden *text*-Elements ist, schreiben sie ein einzelnes Apostroph.

Beispiele für Apostrophe:

```
#FIELDA = 'O''CONNOR'
#FIELDA = 'O"CONNOR'
#FIELDA = "O'CONNOR"
```

In allen drei Fällen erhalten Sie folgende Ausgabe:

```
O'CONNOR
```

Anführungszeichen als Teil eines Textelements benutzen

Es gilt Folgendes, wenn der Natural-Profilparameter `TQ` (Translate Quotation Marks) auf `OFF` gesetzt ist. Die Standardeinstellung ist `ON`.

Für ein Anführungszeichen, das Teil eines in einzelnen Apostrophen stehenden *`text`*-Elements ist, schreiben Sie *ein* Anführungszeichen.

Für ein Anführungszeichen, das Teil eines in Anführungszeichen stehenden *`text`*-Elements ist, schreiben sie *doppelte* Anführungszeichen ("").

Beispiel für Anführungszeichen:

```
#FIELDA = 'O"CONNOR'
#FIELDA = "O""CONNOR"
```

In beiden Fällen erhalten Sie folgende Ausgabe:

```
O"CONNOR
```

Vor einem Feldwert n-mal anzuzeigendes Zeichen definieren — die 'c'(n)-Notation

Soll als Text ein einzelnes Zeichen mehrmals wiederholt werden, verwenden Sie dazu folgende Notation:

```
'c'(n)
```

c steht hierbei für das auszugebende Zeichen, und mit *n* geben Sie an, wie oft das Zeichen generiert werden soll. *n* darf maximal 249 betragen.

Beispiel:

```
WRITE '*'(3)
```

Statt der Apostrophe (') vor und nach dem Zeichen *c* können Sie auch Anführungszeichen (") verwenden.

54 Benutzerkommentare

- Ganze Sourcecode-Zeile als Kommentarzeile benutzen 476
- Teil einer Sourcecode-Zeile als Kommentarzeile benutzen 477

Benutzerkommentare sind zu den Statements des Sourcecodes hinzugefügte oder in ihnen verteilte Beschreibungen oder erläuternde Anmerkungen. Solche Informationen können besonders hilfreich sein, wenn es um das Verstehen und die Pflege von Sourcecode geht, der von einem anderen Programmierer geschrieben oder editiert wurde.

Des Weiteren können die den Anfang eines Kommentars markierenden Zeichen benutzt werden, um die Funktion eines Statements oder mehrere Sourcecode-Zeilen zu Test-Zwecken zeitweilig auszuschalten.

Sie haben in Natural verschiedene Möglichkeiten, im Sourcecode Kommentare einzufügen.

Ganze Sourcecode-Zeile als Kommentarzeile benutzen

Falls Sie eine ganze Sourcecode-Zeile als Kommentarzeile verwenden möchten, geben Sie am Anfang der Zeile Folgendes ein:

- einen Stern und ein Leerzeichen (*),
- zwei Sterne (**) oder
- einen Schrägstrich und einen Stern (/*).

```
*  USER COMMENT
** USER COMMENT
/* USER COMMENT
```

Beispiel:

Wie dem folgenden Beispiel zu entnehmen ist, können Kommentarzeilen auch benutzt werden, um den Sourcecode klar zu strukturieren.

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
```

```
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

Teil einer Sourcecode-Zeile als Kommentarzeile benutzen

Falls Sie nur einen Teil einer Sourcecode-Zeile für einen Kommentar verwenden möchten, geben Sie ein Leerzeichen, einen Schrägstrich und einen Stern (/*) ein; der Rest der Zeile ab dieser Markierung ist damit als Kommentar gekennzeichnet:

```
ADD 5 TO #A         /* USER COMMENT
```

Beispiel:

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

55 Logische Bedingungen

- Einleitung 480
- Relationaler Ausdruck 481
- Erweiterter Relationaler Ausdruck 485
- Auswertung einer logischen Variablen 486
- Felder in logischen Bedingungen 488
- Logische Operatoren in komplexen logischen Ausdrücken 490
- BREAK-Option - Aktuellen Wert mit Wert des vorangegangenen Schleifendurchlaufs vergleichen 491
- IS-Option - Prüfen ob Inhalt von Alphanumerischem oder Unicode-Feld konvertiert werden kann 493
- MASK-Option - Ausgewählte Stellen eines Feldes auf bestimmten Inhalt prüfen 495
- MASK-Option im Vergleich zur IS Option 504
- MODIFIED-Option - Prüfen ob Feldinhalt verändert worden ist 506
- SCAN-Option - Nach einem bestimmten Wert in einem Feld suchen 507
- SPECIFIED-Option - Prüfen ob ein Wert für einen optionalen Parameter übergeben wird 509

Dieses Kapitel beschreibt den Zweck und die Benutzung logischer Bedingungen, die in den folgenden Statements benutzt werden können: FIND, READ, HISTOGRAM, ACCEPT/REJECT, IF, DECIDE FOR, REPEAT

Einleitung

Die Grundform einer logischen Bedingung ist ein relationaler (vergleichender) Ausdruck. Mit den logischen Operatoren AND und OR können mehrere relationale Ausdrücke (AND, OR) zu komplexen logischen Bedingungen verknüpft werden.

Arithmetische Ausdrücke können ebenfalls in logischen Bedingungen verwendet werden.

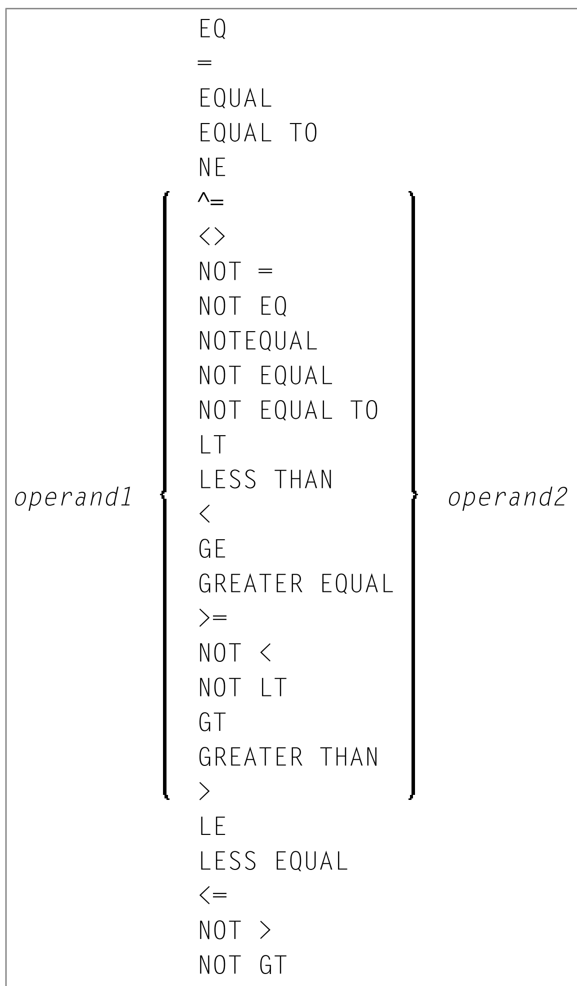
Logische Bedingungen können in den folgenden Statements angegeben werden:

Statement	Bedingungen
FIND	Zusätzlich zu dem primären Selektionskriterium, das in der WITH-Klausel angegeben wird, kann in einer WHERE-Klausel als zusätzliches Selektionskriterium eine logische Bedingung angegeben werden. Die in der WHERE-Klausel angegebene Bedingung wird erst ausgewertet, nachdem ein Datensatz aufgrund des WITH-Kriteriums ausgewählt und gelesen worden ist. In der WITH-Klausel werden „Basic Search Criteria“ (Suchkriterien) angegeben (vgl. FIND-Statement), aber keine logische Bedingung.
READ	In einer WHERE-Klausel kann eine logische Bedingung angegeben werden, die darüber entscheidet, ob ein gerade gelesener Datensatz weiterverarbeitet wird oder nicht. Diese Bedingung wird erst ausgewertet, *nachdem* ein Datensatz gelesen wurde.
HISTOGRAM	In einer WHERE-Klausel kann eine logische Bedingung angegeben werden, die darüber entscheidet, ob ein gerade gelesener Datensatz weiterverarbeitet wird oder nicht. Diese Bedingung wird erst ausgewertet, *nachdem* ein Datensatz gelesen wurde.
ACCEPT/REJECT	Zusätzlich zu den Selektionskriterien, aufgrund derer ein Datensatz mit einem FIND-, READ- oder HISTOGRAM-Statement gelesen wurde, kann in der IF-Klausel eines ACCEPT- oder REJECT-Statements eine weitere logische Bedingung angegeben werden, die über die weitere Verarbeitung eines Datensatzes entscheidet. Diese Bedingung wird erst ausgewertet, nachdem ein Datensatz gelesen wurde und die Verarbeitung des Datensatzes begonnen hat.
IF	Die Ausführung des Statements kann von der Erfüllung einer logischen Bedingung abhängig gemacht werden.
DECIDE FOR	Die Ausführung des Statements kann von der Erfüllung einer logischen Bedingung abhängig gemacht werden.

Statement	Bedingungen
REPEAT	In der UNTIL- oder WHILE-Klausel eines REPEAT-Statements kann eine logische Bedingung angegeben werden, die darüber entscheidet, wann eine Verarbeitungsschleife beendet werden soll.

Relationaler Ausdruck

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	ja	ja
operand2	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	ja	nein

Die obige Operandentabelle ist in der *Statements*-Dokumentation unter *Syntax-Symbole und Operandentabellen* erklärt.

In der Spalte „Mögliche Struktur" der Tabelle steht „E" für arithmetischer Ausdruck, d.h. innerhalb eines relationalen Ausdrucks kann ein beliebiger arithmetischer Ausdruck als Operand verwendet werden. Weitere Informationen siehe `arithmetic-expression` in der Beschreibung des `COMPUTE`-Statements.

Erklärung der Vergleichsoperatoren:

Vergleichsoperator	Erklärung
EQ = EQUAL EQUAL TO	gleich
NE ^= <> NOT = NOT EQ NOTEQUAL NOT EQUAL NOT EQUAL TO	ungleich
LT LESS THAN <	kleiner als
GE GREATER EQUAL >=	größer als oder gleich
NOT < NOT LT	nicht größer als
GT GREATER THAN >	größer als
LE LESS EQUAL <=	kleiner als oder gleich less

Vergleichsoperator	Erklärung
NOT > NOT GT	nicht größer als

Beispiele für relationale Ausdrücke:

```
IF NAME = 'SMITH'
IF LEAVE-DUE GT 40
IF NAME = #NAME
```

Weitere Informationen über den Vergleich von Arrays in einem relationalen Ausdruck siehe ***Verarbeitung von Arrays***.

Anmerkung: Wird ein Gleitkomma-Operand verwendet, so erfolgt der Vergleich in Gleitkommaform. Da **Gleitkomma-Zahlen** per se nur eine begrenzte Genauigkeit haben, lassen sich Rundungs- bzw. Abschneidefehler bei der Konvertierung von Zahlen in/aus Gleitkommaform nicht ausschließen.

Arithmetische Ausdrücke in logischen Bedingungen

Das folgende Beispiel zeigt, wie arithmetische Ausdrücke in logischen Bedingungen eingesetzt werden können:

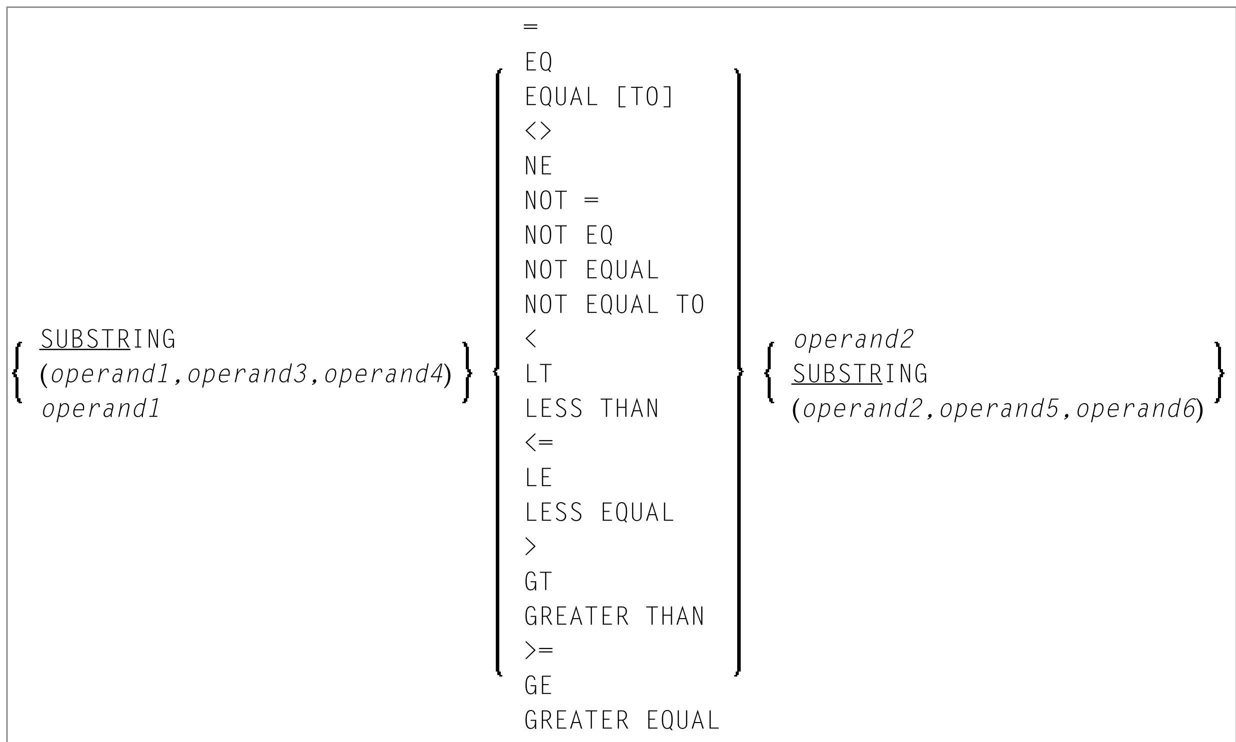
```
IF #A + 3 GT #B - 5 AND #C * 3 LE #A + #B
```

Handles in logischen Bedingungen

Wenn die Operanden in einem relationalen Ausdruck Handles sind, dürfen nur `EQUAL`- und `NOT EQUAL`-Operatoren verwendet werden.

SUBSTRING-Option in relationalem Ausdruck

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U					B							ja	ja
operand2	C	S	A		N		A	U					B							ja	nein
operand3	C	S							N	P	I		B							ja	nein
operand4	C	S							N	P	I									ja	nein
operand5	C	S							N	P	I									ja	nein
operand6	C	S							N	P	I									ja	nein

Mit der `SUBSTRING`-Option können Sie einen *Teil* eines alphanumerischen, binären oder Unicode-Feldes vergleichen. Nach dem Feldnamen (*operand1*) geben Sie zunächst die erste Stelle (*operand3*) und dann die Länge (*operand4*) des zu vergleichenden Feldteils an.

Sie können auch einen Feldwert mit einem Teil eines anderen Feldwertes vergleichen. Nach dem Feldnamen (*operand2*) geben Sie zunächst die erste Stelle (*operand5*) und dann die Länge (*operand6*) des Feldteils an, mit dem *operand1* verglichen werden soll.

Sie können auch beide Formen miteinander kombinieren, d.h. `SUBSTRING` gleichzeitig für *operand1* und für *operand2* angeben.

Beispiele:

Dieser Ausdruck vergleicht die 5. bis einschließlich 12. Stelle des Wertes in Feld #A mit dem Wert von Feld #B:

```
SUBSTRING(#A,5,8) = #B
```

wobei 5 die erste Stelle und 8 die Länge ist.

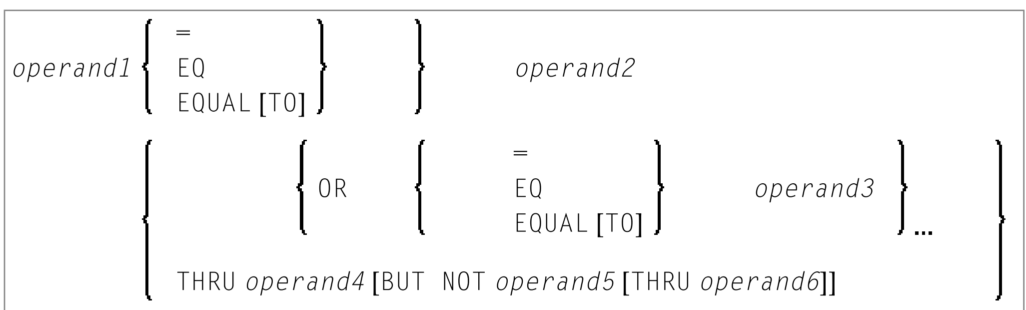
Dieser Ausdruck vergleicht den Wert von Feld #A mit der 3. bis einschließlich 6. Stelle des Wertes in Feld #B:

```
#A = SUBSTRING(#B,3,4)
```

Anmerkung: Wenn Sie *operand3*/*operand5* weglassen, wird ab Anfang des Feldes verglichen. Wenn Sie *operand4*/*operand6* weglassen, wird ab der angegebenen Stelle (*operand3*/*operand5*) bis zum Ende des Feldes verglichen.

Erweiterter Relationaler Ausdruck

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand2	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand3	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand4	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand5	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand6	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein

* Mathematische Funktionen und Systemvariablen sind erlaubt. Gruppenwechsel-Funktionen sind nicht erlaubt.

Operand3 kann auch unter Verwendung einer `MASK`- oder `SCAN`-Option angegeben werden; d.h. er kann angegeben werden als:

```
MASK (mask-definition) [operand]
MASK operand
SCAN operand
```

Einzelheiten zu diesen Optionen finden Sie unter ***MASK-Option*** und ***SCAN-Option***.

Beispiele:

```
IF #A = 2 OR = 4 OR = 7
IF #A = 5 THRU 11 BUT NOT 7 THRU 8
```

Auswertung einer logischen Variablen

Syntax:

```
operand1
```

Diese Option kann in Verbindung mit einer logischen Variablen (Format L) eingesetzt werden. Eine logische Variable kann die Werte `TRUE` (wahr) oder `FALSE` (falsch) haben. Mit *operand1* geben Sie den Namen der Variablen an.

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A													L				nein	nein

Beispiel für eine logische Variable:

```
** Example 'LOGICX05': Logical variable in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #SWITCH (L)  INIT <true>
1 #INDEX  (I1)
END-DEFINE
*
FOR #INDEX 1 5
  WRITE NOTITLE #SWITCH (EM=FALSE/TRUE) 5X 'INDEX =' #INDEX
  WRITE NOTITLE #SWITCH (EM=OFF/ON) 7X 'INDEX =' #INDEX
  IF #SWITCH
    MOVE FALSE TO #SWITCH
  ELSE
    MOVE TRUE TO #SWITCH
  END-IF
  /*
  SKIP 1
END-FOR
END
```

Ausgabe des Programms `LOGICX05`:

```
TRUE     INDEX =    1
ON       INDEX =    1

FALSE    INDEX =    2
OFF      INDEX =    2

TRUE     INDEX =    3
ON       INDEX =    3

FALSE    INDEX =    4
OFF      INDEX =    4

TRUE     INDEX =    5
ON       INDEX =    5
```

Felder in logischen Bedingungen

Bei der Konstruktion logischer Bedingungen dürfen sowohl Datenbankfelder als auch Benutzervariablen verwendet werden. Datenbankfelder, die Teil einer Periodengruppe oder multiple Felder sind, dürfen ebenfalls verwendet werden. Wenn ein Bereich von Werten für multiple Felder oder ein Bereich von Ausprägungen für Periodengruppen angegeben wird, dann ist die Bedingung erfüllt, wenn der Suchwert in einem Wert bzw. einer Ausprägung innerhalb des angegebenen Bereichs gefunden wird.

Jeder verwendete Wert muss mit dem ihm in einem relationalen Ausdruck gegenüberstehenden Feld kompatibel sein. Dezimalstellen können nur bei Werten für numerische Felder angegeben werden, wobei die Anzahl der Dezimalstellen von Wert und Feld kompatibel sein muss.

Haben zwei Operanden unterschiedliches Format, wird das Format des zweiten Operanden in das des ersten umgesetzt.

Anmerkung: Eine numerische Konstante ohne Dezimalzeichen-Notation wird im Wertebereich `-2147483648` bis `+2147483647` im Format I gespeichert, siehe ***Numerische Konstante***. Deshalb wird beim Vergleich mit einer solchen ganzzahligen Konstante der *`operand2`* in einen ganzzahligen Wert umgesetzt. Das hat zur Folge, dass die Nachkommastellen beim *`operand2`* abgeschnitten und nicht berücksichtigt werden.

Beispiel:

```
 IF 0 = 0.5    /* is true because 0.5 (operand2) is converted to 0 (format I of
operand1)
 IF 0.0 = 0.5  /* is false
 IF 0.5 = 0    /* is false
 IF 0.5 = 0.0  /* is false
```

Die folgende Tabelle zeigt, welche Operandenformate zusammen in einer logischen Bedingung verwendet werden können:

operand1	*operand2*												
	A	**U**	**B***n* (*n*=<4)	**B***n* (*n*>=5)	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
A	Y	Y	Y	Y									
U	Y	Y	[2]	[2]									
B*n* (*n*=<4)	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		
B*n* (*n*>=5)	Y	Y	Y	Y									

operand1	*operand2*												
	A	**U**	**B*n* (*n*=<4)**	**B*n* (*n*>=5)**	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
D			Y		Y	Y	Y	Y		Y	Y		
T			Y		Y	Y	Y	Y		Y	Y		
I			Y		Y	Y	Y	Y		Y	Y		
F			Y		Y	Y	Y	Y		Y	Y		
L													
N			Y		Y	Y	Y	Y		Y	Y		
P			Y		Y	Y	Y	Y		Y	Y		
GH [1]												Y	
OH [1]													Y

Legende:

Y = ja

[1] GH = GUI Handle, OH = Object Handle.

[2] Es wird davon ausgegangen, dass der Binärwert Unicode-Codepunkte enthält, und der Vergleich wird wie für einen Vergleich zweier Unicode-Werte durchgeführt. Die Länge des binären Feldes muss geradzahlig sein.

Wird ein Array mit einem Skalarwert in Relation gesetzt, so wird jedes Element des Arrays mit dem Skalarwert verglichen; die Bedingung ist erfüllt, wenn mindestens ein Array-Element die Bedingung erfüllt (`ODER`-Verknüpfung).

Wird ein Array mit einem Array in Relation gesetzt, so wird jedes Element des einen Arrays mit dem entsprechenden Element des anderen Arrays verglichen; die Bedingung ist nur dann erfüllt, wenn alle Element-Vergleiche die Bedingung erfüllen (`UND`-Verknüpfung).

Siehe auch *Verarbeitung von Arrays*.

Anmerkung: Phonetische Deskriptoren (Adabas) dürfen in einer logischen Bedingung nicht verwendet werden

Beispiele für logische Bedingungen:

```
FIND EMPLOYEES-VIEW WITH CITY = 'BOSTON' WHERE SEX = 'M'
READ EMPLOYEES-VIEW BY NAME WHERE SEX = 'M'
ACCEPT IF LEAVE-DUE GT 45
IF #A GT #B THEN COMPUTE #C = #A + #B
REPEAT UNTIL #X = 500
```

Logische Operatoren in komplexen logischen Ausdrücken

Mittels der Boole'schen Operatoren AND, OR und NOT ist es möglich, logische Bedingungen miteinander zu verknüpfen. Mit Hilfe von Klammern können logische Bedingungen logisch zusammengefasst werden.

Die Operatoren werden in der folgenden Reihenfolge ausgewertet:

Priorität	Logische Verknüpfung	Bedeutung
1	( )	Klammer-Rechnung
2	NOT	Negation
3	AND	UND-Verknüpfung
4	OR	ODER-Verknüpfung

Die folgenden logischen Bedingungen können miteinander verknüpft werden, um einen komplexen logischen Ausdruck zu bilden:

- **Relationale Ausdrücke**
- **Erweiterte relationale Ausdrücke**
- MASK**-Option**
- SCAN**-Option**
- BREAK**-Option**

Die Syntax für eine logische Bedingung (*logical-expression*) ist wie folgt:

```
[NOT] { logical-condition-criterion } [ { OR  } logical-expression ] ...
      { (logical-expression)        }   { AND }
```

Beispiele für `logical-expressions`**:**

```
FIND STAFF-VIEW WITH CITY = 'TOKYO'
     WHERE BIRTH GT 19610101 AND SEX = 'F'
IF NOT (#CITY = 'A' THRU 'E')
```

Informationen über den Vergleich von Arrays in einem logischen Ausdruck finden Sie in ***Verarbeitung von Arrays***.

Anmerkung: Wenn mehrere `logical-condition-criteria` mit `AND` verknüpft werden, wird die Auswertung beendet, sobald das erste dieser Kriterien gefunden wird, das nicht erfüllt ist.

BREAK-Option - Aktuellen Wert mit Wert des vorangegangenen Schleifendurchlaufs vergleichen

Mit der `BREAK`-Option kann der aktuelle Wert eines Feldes (oder eines Teils eines Feldes) mit dem Wert verglichen werden, den das Feld im vorangegangenen Durchlauf durch die Verarbeitungsschleife hatte.

Syntax:

```
BREAK [OF] operand1 [/n/]
```

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1		S					A	U	N	P	I	F	B	D	T	L				ja	nein

Syntax-Elementbeschreibung:

operand1	Mit *operand1* geben Sie das Feld an, das überprüft werden soll. Eine bestimmte Ausprägung eines Arrays kann auch als ein Kontrollfeld benutzt werden.
/n/	Soll nur ein Teil des Feldes überprüft werden, so geben Sie eine mit Schrägstrichen eingegrenzte Zahl *n* an, die angibt, wieviele Stellen (von links nach rechts gezählt) des Feldes auf einen Wertwechsel überprüft werden sollen. Die Notation */n/* kann nur bei Operanden des Formats A, B, N oder P benutzt werden. Eine BREAK-Bedingung ist erfüllt, wenn der Wert des Kontrollfeldes (bzw. der angegebenen Stellen des Feldes) sich ändert. Eine BREAK-Bedingung ist nicht erfüllt, wenn eine AT END OF DATA-Bedingung auftritt. Beispiel: In diesem Beispiel wird überprüft, ob der Wert der ersten Stelle des Feldes FIRST-NAME sich geändert hat. `BREAK FIRST-NAME /1/` Der Einsatz von Natural-Systemfunktionen (die bei einem AT BREAK-Statement zur Verfügung stehen) ist bei der BREAK-Option nicht erlaubt.

Beispiel für BREAK-Option:

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

Ausgabe des Programms `LOGICX03`:

```
  DATE            NAME               FIRST-NAME
   OF
 BIRTH
---------- -------------------- --------------------

1940-01-01 GARRET               WILLIAM
1940-01-09 TAILOR               ROBERT
1940-01-09 PIETSCH              VENUS
1940-01-31 LYTTLETON            BETTY

--------------------------------------------------

1940-02-02 WINTRICH             MARIA
1940-02-13 KUNEY                MARY
1940-02-14 KOLENCE              MARSHA
1940-02-24 DILWORTH             TOM

--------------------------------------------------

1940-03-03 DEKKER               SYLVIA
1940-03-06 STEFFERUD            BILL
```

IS-Option - Prüfen ob Inhalt von Alphanumerischem oder Unicode-Feld konvertiert werden kann

Syntax:

operand1 IS (*format*)

Mit dieser Option können Sie prüfen, ob der Inhalt eines alphanumerischen oder Unicode-Feldes (*operand1*) in ein bestimmtes anderes Format übertragbar ist.

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U												ja	nein

Dieses *Format* kann sein:

N*ll.ll*	Numerisch mit Länge *ll.ll*.
F*ll*	Gleitkomma (floating point) mit Länge *ll*.
D	Datum. Folgende Datumsformate sind möglich: *dd-mm-yy*, *dd-mm-yyyy*, *ddmmyyyy* (*dd* = Tag, *mm* = Monat, *yy* oder *yyyy* = Jahr). Die Abfolge der Tages-, Monats- und Jahreskomponenten sowie die Trennzeichen zwischen den Komponenten werden durch den Profilparameter DTFORM (der in der *Parameter-Referenz*-Dokumentation beschrieben ist) bestimmt.
T	Zeit (Time); entsprechend dem Standard-Zeitanzeigeformat.
P*ll.ll*	Gepackt numerisch mit Länge *ll.ll*.
I*ll*	Ganzzahl (integer) mit Länge *ll*.

Bei der Prüfung werden für den Wert in *operand1* vor- oder nachgestellte Leerzeichen ignoriert.

Die Prüfung mit der IS-Option ist sinnvoll, wenn beispielsweise vor Ausführung der mathematischen Funktion VAL (Erhalt des numerischen Wertes eines alphanumerischen Feldes) das Format des Wertes überprüft wird, um zu vermeiden, dass ein falsches Format einen Laufzeitfehler verursacht.

Anmerkung: Mit der IS-Option kann nicht geprüft werden, ob der Wert eines alphanumerischen Feldes in dem angegebenen Format ist, sondern ob er in das Format *übertragbar* ist. Um zu prüfen, ob ein Wert in einem bestimmten Format ist, können Sie die ***MASK-Option*** verwenden.

Beispiel für IS-Option:

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
```

```
            MARK *#FIELDA
  END-IF
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

Ausgabe des Programms LOGICX04:

```
#DATE 150487    #FIELDA










INPUT IS NOT IN DATE FORMAT (YY-MM-DD)
```

Weitere Informationen siehe ***MASK-Option im Vergleich zur IS-Option***.

MASK-Option - Ausgewählte Stellen eines Feldes auf bestimmten Inhalt prüfen

Mit der MASK-Option können Sie bestimmte ausgewählte Stellen eines Feldes nach einem bestimmten Wert absuchen.

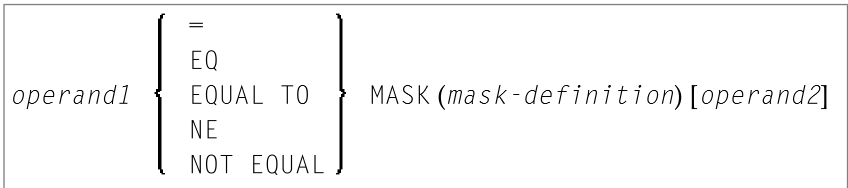
Folgende Themen werden behandelt:

- Konstante Maske
- Variable Maske
- Zeichen in einer Maske
- Maskenlänge
- Datumsprüfungen
- Prüfung unter Verwendung von Konstanten oder Variablen
- Bereichsprüfungen

- Auf gepackte oder ungepackte numerische Daten abprüfen

Konstante Maske

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U	N	P										ja	nein
operand2	C	S					A	U	N	P			B							ja	nein

Operand2 kann nur angegeben werden, wenn die *mask-definition* mindestens ein X enthält. *Operand1* und *operand2* müssen formatkompatibel sein:

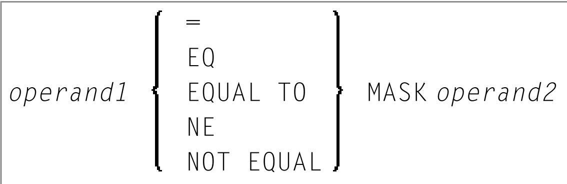
- wenn *operand1* Format A hat, muss *operand2* Format A, B, N oder U haben
- wenn *operand1* Format U hat, muss *operand2* Format A, B, N oder U haben
- wenn *operand1* Format N oder P hat, muss *operand2* Format N oder P haben.

Wird ein X in der *mask-definition* angegeben, werden die betreffenden inhaltlichen Stellen von *operand1* und *operand2* zum Wertevergleich ausgewählt.

Variable Maske

Anstatt einer konstanten *mask-definition* (siehe oben) können Sie auch eine variable MASK-Definition angeben:

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U	N	P										yes	no
operand2		S					A	U												yes	no

Der Inhalt von `operand2` wird dann als `MASK`-Definition genommen. Nachgestellte Leerzeichen in `operand2` werden ignoriert.

- Wenn `operand1` Format A, N oder P hat, muss `operand2` Format A haben.
- Wenn `operand1` Format U hat, muss `operand2` Format A haben.

Zeichen in einer Maske

In einer `mask-definition` können Sie folgende Zeichen verwenden (die Masken-Definition ist bei einer konstanten Maske in der `mask-definition` und bei einer variablen Maske in `operand2` enthalten):

Zeichen	Bedeutung
. oder ? oder _	Ein Punkt (.) oder Fragezeichen (?) oder ein Unterstrich (_) markiert eine einzelne Stelle, die nicht überprüft werden soll.
* oder %	Eine beliebige Anzahl von Stellen, die nicht überprüft werden sollen.
/	(Schrägstrich) Prüft, ob der Wert mit einem (oder mehreren) bestimmten Zeichen endet. Beispiel: Die folgende Bedingung ist wahr, wenn entweder an der letzten Stelle des Feldes ein `E` steht oder nach dem `E` nur noch Leerzeichen stehen:

Zeichen	Bedeutung
	`IF #FIELD = MASK (*'E'/)`
A	Eine Stelle, die nach Groß- oder Kleinbuchstaben abgesucht werden soll.
'*c*'	Eine oder mehrere Stellen, die nach den in Apostrophen (') stehenden Zeichen abgesucht werden sollen (doppelte Apostrophe bedeuten, dass innerhalb der Zeichenkette nach einem Apostroph gesucht wird). Wenn *operand1* Unicode-Format hat, muss '*c*' Unicode-Zeichen enthalten.
C	Eine Stelle, die nach alphanumerischem Inhalt (Groß- oder Kleinbuchstabe, Zahl) oder Leerzeichen abgesucht werden soll.
DD	Zwei Stellen, die nach einem gültigen Tagesdatum abgesucht werden sollen (01 - 31; abhängig von den Werten für MM und YY/YYYY, falls angegeben; siehe auch ***Datumsprüfungen***).
H	Eine Stelle, die nach einem Hexadezimalzeichen (A - F, 0 - 9) abgesucht werden soll.
JJJ	Die Stellen sollen nach einem gültigen Julianischen Tag abgesucht werden, d.h. die Tageszahl im Jahr: 001-366, abhängig vom Wert von YY/YYYY, wenn angegeben; siehe auch ***Datumsprüfungen***.
L	Eine Stelle, die nach Kleinbuchstaben (a - z) abgesucht werden soll.
MM	Zwei Stellen, die nach einer gültigen Monatsangabe (01 – 12) abgesucht werden sollen; siehe auch ***Datumsprüfungen***.
N	Eine Stelle, die nach einer Ziffer abgesucht werden soll.
n...	Eine oder mehrere Stellen, die nach einem numerischen Wert im Bereich von 0 bis *n* abgesucht werden sollen.
n1-*n2* oder *n1*:*n2*	Stellen, die nach einem numerischen Wert im Bereich von *n1*-*n2* abgesucht werden sollen. *n1* und *n2* müssen gleich lang sein.
P	Eine Stelle, die nach einem druckbaren Zeichen (U, L, N oder S – Buchstabe, Zahl oder Sonderzeichen) abgesucht werden soll.
S	Eine Stelle, die nach Sonderzeichen abgesucht werden soll. Siehe auch *Support of Different Character Sets with NATCONV.INI* in der *Operations*-Dokumentation.
U	Eine Stelle, die nach Großbuchstaben (A – Z) abgesucht werden soll.
X	Eine Stelle, die mit der entsprechenden Stelle des auf die *mask-definition* folgenden Wertes (*operand2*) verglichen werden soll. In einer variablen Maske ist X nicht erlaubt, da sinnlos.
YY	Zwei Stellen, die nach einer gültigen Jahreszahl (00 - 99) abgesucht werden sollen; siehe auch ***Datumsprüfungen***.
YYYY	Vier Stellen, die nach einer gültigen Jahreszahl (0000-2699) abgesucht werden sollen.
Z	Eine Stelle, die nach einem Zeichen abgesucht werden soll, dessen linkes Halbbyte hexadezimal 3 oder 7 und dessen rechtes Halbbyte hexadezimal 0 – 9 ist. Damit können Sie korrekt nach Ziffern in negativen Zahlen suchen. Mit N (womit Sie eine Stelle nach einer Ziffer absuchen können) erhalten Sie bei Suche von Ziffern in negativen

Zeichen	Bedeutung
	Zahlen falsche Ergebnisse, da das Vorzeichen in der letzten Stelle der Zahl gespeichert ist, wodurch diese Stelle hexadezimal als Nicht-Ziffer dargestellt wird. Geben Sie innerhalb einer Maske für jede Reihe numerischer Stellen, die geprüft werden sollen, nur jeweils ein Z an.

Maskenlänge

Welche Stellen abgesucht werden sollen, ergibt sich aus der Definition der Maske.

Beispiel:

```
DEFINE DATA LOCAL
1 #CODE (A15)
END-DEFINE
...
IF #CODE = MASK (NN'ABC'....NN)
...
```

Die ersten beiden Stellen von #CODE werden nach einem numerischen Wert abgesucht, die 3. bis 5. Stelle nach dem Wert ABC, die 10. und 11. Stelle nach einem numerischen Wert; die 6. bis 9. und 12. bis 15. Stelle werden nicht überprüft.

Datumsprüfungen

Pro Maske darf nur ein Datum geprüft werden. Wenn in der Maske derselbe Datumsbestandteil (JJJ, DD, MM, YY or YYYY) mehr als einmal angegeben wird, dann wird nur der Wert der letzten Ausprägung auf Konsistenz mit anderen Datumsbestandteilen geprüft.

Wird bei der Prüfung eines Tagesdatums (DD) keine Monatsangabe (MM) in der Maske gemacht, wird der aktuelle Monat angenommen.

Wird bei der Prüfung eines Tagesdatums (DD) oder Julianischen Tagesdatums (JJJ) keine Jahresangabe (YY bzw. YYYY) in der Maske gemacht, wird das aktuelle Jahr angenommen.

Bei der Prüfung einer zweistelligen Jahreszahl (YY) wird das aktuelle Jahrhundert angenommen, sofern kein Sliding Window oder Fixed Window gesetzt ist. Weitere Einzelheiten über Sliding oder Fixed Windows, entnehmen Sie dem Profilparameter YSLW in der *Parameter Reference*-Dokumentation.

Beispiel 1:

```
MOVE 1131 TO #DATE (N4)
IF #DATE = MASK (MMDD)
```

In diesem Beispiel werden Monat und Tag auf ihre Gültigkeit überprüft. Der Monatswert `11` ist gültig, wohingegen der Tageswert `31` ungültig ist, da der 11. Monat nur 30 Tage hat.

Beispiel 2:

```
IF #DATE(A8) = MASK (MM'/'DD'/'YY)
```

In diesem Beispiel wird überprüft, ob das Feld `#DATE` ein gültiges Datum im Format `MM/DD/YY` (Monat/Tag/Jahr) enthält.

Beispiel 3:

```
IF #DATE (A8) = MASK (1950-2020MMDD)
```

In diesem Beispiel wird der Inhalt des Feldes `#DATE` auf eine vierstellige Zahl im Bereich 1950 bis 2020 geprüft, auf die ein gültiger Monat und Tag im aktuellen Jahr folgen:

Anmerkung: Obwohl es so aussieht, ermöglicht die oben angegebene Maske nicht das Abprüfen auf ein gültiges Datum in den Jahren 1950 - 2020, weil der numerische Wertebereich 1950-2020 unabhängig von der Gültigkeitsprüfung für Monat und Tag abgeprüft wird. Die Prüfung liefert die beabsichtigten Ergebnisse mit Ausnahme des 29. Februars, denn an diesem Tag ist das Ergebnis davon abhängig, ob das aktuelle Jahr ein Schaltjahr ist oder nicht. Um zusätzlich zur Datumsgültigkeitsprüfung auf einen bestimmten Bereich von Jahren zu prüfen, bauen Sie eine Gültigkeitsprüfung für das Datum und eine andere für den Bereich in Ihrem Programm ein.

```
IF #DATE (A8) = MASK (YYYYMMDD) AND #DATE = MASK (1950-2020)
```

Beispiel 4:

```
IF #DATE (A4) = MASK (19-20YY)
```

In diesem Beispiel wird überprüft, ob das Feld `#DATE` eine zweistellige Zahl im Bereich von `19` bis `20`, gefolgt von einem gültigen zweistelligen Jahr (`00` bis `99`) enthält. Das Jahrhundert wird wie oben beschrieben von Natural angegeben.

Anmerkung: Obwohl es so aussieht, ermöglicht die oben angegebene Maske nicht das Abprüfen auf ein gültiges Jahr im Bereich von `1900` bis `2099`, weil der numerische Wertebereich `19 - 20` unabhängig von der Gültigkeitsprüfung für das Jahr abgeprüft wird. Um auf Bereiche von Jahren abzuprüfen, bauen Sie eine Gültigkeitsprüfung für das Datum und eine andere für den Bereich in Ihrem Programm ein:

```
IF #DATE (A10) = MASK (YYYY'-'MM'-'DD) AND #DATE = MASK (19-20)
```

Prüfung unter Verwendung von Konstanten oder Variablen

Ist der für die Maskenprüfung verwendete Wert eine Konstante oder der Inhalt einer Variablen, dann muss dieser Wert (*operand2*) unmittelbar nach der *mask-definition* angegeben werden.

operand2 muss mindestens so lang sein wie die Maske.

In der Maske geben Sie für jede zu überprüfende Stelle ein `X` und für jede nicht zu überprüfende Stelle einen Punkt (.) (oder `?` oder _) an.

Beispiel:

```
DEFINE DATA LOCAL
1 #NAME (A15)
END-DEFINE
...
IF #NAME = MASK (..XX) 'ABCD'
...
```

Hier wird geprüft, ob die 3. bis 4. Stelle des Feldes `#NAME` den Wert `CD` enthält. Die ersten beiden Stellen werden nicht überprüft.

Wieviele Stellen geprüft werden, hängt von der Länge der definierten Maske ab. Die Maske wird immer linksbündig auf das zu überprüfende Feld bzw. die zu prüfende Konstante ausgerichtet. *Operand1* muss dasselbe Format haben wie *operand2*.

Hat das zu überprüfende Feld (*operand1*) Format A, muss ein konstanter Wert (*operand2*) in Apostrophen stehen. Ist das Feld numerisch, muss der Wert eine Zahl oder der Inhalt eines numerischen Datenbankfeldes bzw. einer numerischen Benutzervariablen sein.

In jedem Fall werden Zeichen/Stellen, die nicht an einer in der Maskendefinition mit `X` markierten Stelle stehen, ignoriert.

Die `MASK`-Bedingung ist erfüllt, wenn alle in der Maske angegebenen Stellen dem geforderten Wert entsprechen.

Beispiel:

```
** Example 'LOGICX01': MASK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
HISTOGRAM EMPLOY-VIEW CITY
  IF CITY =
MASK (....XX) '....NN'


    DISPLAY NOTITLE CITY *NUMBER
  END-IF
END-HISTOGRAM
*
END
```

In diesem Beispielprogramm werden nur Datensätze akzeptiert, bei denen das Feld `CITY` einen Wert enthält, der an der 5. und 6. Stelle jeweils ein `N` hat.

Bereichsprüfungen

Bei Bereichsprüfungen wird die Anzahl der verifizierten Stellen durch die Genauigkeit des in der Maske angegebenen Wertes definiert. Zum Beispiel würde die Maske `(...193...)` die Stellen 4 bis 6 nach einer dreistelligen Zahl im Bereich von `000` bis `193` überprüfen.

Weitere Beispiele für Masken-Definitionen:

- In diesem Beispiel wird überprüft, ob alle Stellen des Feldes `#NAME` einen Buchstaben enthalten:

  ```
  IF #NAME (A10) = MASK (AAAAAAAAAA)
  ```

- In diesem Beispiel wird überprüft, ob die 4. bis 6. Stelle von `#NUMBER` eine Zahl enthält:

  ```
  IF #NUMBER (A6) = MASK (...NNN)
  ```

- In diesem Beispiel wird überprüft, ob die 4. - 6. Stelle von #VALUE den Wert 123 enthält:

```
IF #VALUE(A10) = MASK (...'123')
```

- In diesem Beispiel wird überprüft, ob das Nummernschild-Feld #LICENSE ein KFZ-Kennzeichen enthält, das mit NY- beginnt, gefolgt vom Wert der letzten fünf Stellen des Feldes #VALUE:

```
DEFINE DATA LOCAL
1 #VALUE(A8)
1 #LICENSE(A8)
END-DEFINE
INPUT 'ENTER KNOWN POSITIONS OF LICENSE PLATE:' #VALUE
IF #LICENSE = MASK ('NY-'XXXXX) #VALUE
```

- Die folgende Bedingung wird von jedem Wert erfüllt, der NAT und AL enthält, ganz gleich wieviele andere Zeichen zwischen NAT und AL stehen (dies würde z.B. auf die Werte NATURAL und NATIONALITAET genauso zutreffen wie auf den Wert NATAL):

```
MASK('NAT'*'AL')
```

Auf gepackte oder ungepackte numerische Daten abprüfen

In Altanwendungen sind gepackte oder ungepackte numerische Variablen häufig mit alphanumerischen oder binären Feldern neu definiert. Solche Neudefinitionen sind nicht empfehlenswert, weil die Verwendung einer gepackten oder ungepackten Variablen in einer Zuweisung oder Berechnung zu Fehlern oder unvorhersagbaren Ergebnissen führen kann.

Um den Inhalt einer solchen neu definierten Variablen auf Gültigkeit hin abzuprüfen, bevor die Variable verwendet wird, benutzen Sie die Option N (siehe *Zeichen in einer Maske*) so oft wie die Anzahl der Stellen minus 1 mal, gefolgt von einer einzelnen Option Z.

Beispiele :

```
IF #P1 (P1) = MASK (Z)
IF #N4 (N4) = MASK (NNNZ)
IF #P5 (P5) = MASK (NNNNZ)
```

Weitere Informationen zum Abprüfen von Feldinhalten siehe *MASK-Option im Vergleich zur IS-Option*.

MASK-Option im Vergleich zur IS Option

Dieser Abschnitt beschreibt den Unterschied zwischen der `MASK`-Option und der `IS`-Option und enthält ein Beispielprogramm, das den Unterschied zwischen den beiden Optionen veranschaulicht.

Mit der IS-Option können Sie prüfen, ob der Inhalt eines alphanumerischen oder Unicode-Felds in ein bestimmtes anderes Format umgesetzt werden kann. Sie können mit dieser Option jedoch nicht abprüfen, ob der Wert eines alphanumerischen Feldes im angegeben Format vorliegt.

Mit der `MASK`-Option können Sie den Inhalt einer neu definierten gepackten oder ungepackten numerischen Variablen auf Gültigkeit abprüfen.

Beispiel zur Erläuterung des Unterschieds:

```
** Example 'LOGICX09': MASK versus IS option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #A2   (A2)
1 REDEFINE #A2
  2 #N2 (N2)
1 REDEFINE #A2
  2 #P3 (P3)
1 #CONV-N2 (N2)
1 #CONV-P3 (P3)
END-DEFINE
*
#A2 := '12'
WRITE NOTITLE   'Assignment #A2 := "12" results in:'
PERFORM SUBTEST
#A2 := '-1'
WRITE NOTITLE / 'Assignment #A2 := "-1" results in:'
PERFORM SUBTEST
#N2 :=  12
WRITE NOTITLE / 'Assignment #N2 :=  12  results in:'
PERFORM SUBTEST
#N2 :=  -1
WRITE NOTITLE / 'Assignment #N2 :=  -1  results in:'
PERFORM SUBTEST
#P3 :=  12
WRITE NOTITLE / 'Assignment #P3 :=  12  results in:'
PERFORM SUBTEST
#P3 :=  -1
WRITE NOTITLE / 'Assignment #P3 :=  -1  results in:'
PERFORM SUBTEST
*

DEFINE SUBROUTINE SUBTEST
IF #A2 IS (N2) THEN
```

```
  #CONV-N2 := VAL(#A2)
  WRITE NOTITLE 12T '#A2 can be converted to' #CONV-N2 '(N2)'
END-IF
IF #A2 IS (P3) THEN
  #CONV-P3 := VAL(#A2)
  WRITE NOTITLE 12T '#A2 can be converted to' #CONV-P3 '(P3)'
END-IF
IF #N2 = MASK(NZ) THEN
  WRITE NOTITLE 12T '#N2 contains the valid unpacked number' #N2
END-IF
IF #P3 = MASK(NNZ) THEN
  WRITE NOTITLE 12T '#P3 contains the valid packed number' #P3
END-IF
END-SUBROUTINE
*
END
```

Ausgabe des Programms `LOGICX09`:

```
Assignment #A2 := '12' results in:
           #A2 can be converted to  12 (N2)
           #A2 can be converted to   12 (P3)
           #N2 contains the valid unpacked number  12

Assignment #A2 := '-1' results in:
           #A2 can be converted to  -1 (N2)
           #A2 can be converted to   -1 (P3)

Assignment #N2 :=  12  results in:
           #A2 can be converted to  12 (N2)
           #A2 can be converted to   12 (P3)
           #N2 contains the valid unpacked number  12

Assignment #N2 :=  -1  results in:
           #N2 contains the valid unpacked number  -1

Assignment #P3 :=  12  results in:
           #P3 contains the valid packed number   12

Assignment #P3 :=  -1  results in:
           #P3 contains the valid packed number   -1
```

MODIFIED-Option - Prüfen ob Feldinhalt verändert worden ist

Syntax:

```
operand1 [NOT] MODIFIED
```

Mit dieser Option wird überprüft, ob der Inhalt eines Feldes während der Ausführung eines `INPUT`- oder `PROCESS PAGE`-Statements verändert worden ist. Als Voraussetzung muss dem Feld eine Kontrollvariable mit dem Parameter `CV` zugewiesen worden sein.

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1		S	A														C			nein	nein

Von einem `INPUT`- oder `PROCESS PAGE`-Statement referenzierte Kontrollvariablen erhalten immer den Status „not modified“ (nicht verändert), wenn die Map zur Ausgabe an das Terminal übertragen wird.

Wird der Inhalt eines Feldes, das eine Kontrollvariable (*operand1*) referenziert, verändert, erhält die Kontrollvariable den Status „modified“ (verändert). Referenzieren mehrere multiple Felder dieselbe Kontrollvariable, so erhält die Variable den Status `MODIFIED`, wenn mindestens eines dieser Felder verändert wurde.

Ist die Kontrollvariable *operand1* ein Array, so erhält die Variable den Status „modified“, wenn mindestens eins der Elemente des Arrays verändert wurde (`ODER`-Verknüpfung).

Beispiel für MODIFIED-Option:

```
** Example 'LOGICX06': MODIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A    (A1)
1 #B    (A1)
END-DEFINE
*
MOVE (AD=I) TO #ATTR
*
INPUT (CV=#ATTR) #A #B
IF #ATTR NOT MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS NOT BEEN MODIFIED'
END-IF
```

```
*
IF #ATTR MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS BEEN MODIFIED'
END-IF
*
END
```

Ausgabe des Programms `LOGICX06`:

```
#A    #B
```

Nach Eingabe eines Wertes und Drücken der Freigabetaste, wird die folgende Ausgabe angezeigt:

```
FIELD #A OR #B HAS BEEN MODIFIED
```

SCAN-Option - Nach einem bestimmten Wert in einem Feld suchen

Syntax:

```
           { =         }
           { EQ        }          { operand2   }
operand1   { EQUAL TO  }  SCAN    { (operand2) }
           { NE        }
           { NOT EQUAL }
```

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate													Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U	N	P										ja	nein
operand2	C	S					A	U					B*							ja	nein

* *Operand2* darf nur binär sein, wenn *operand1* alphanumerisch oder Unicode ist. Wenn *operand1* Format U hat und *operand2* Format B, dann muss die Länge von *operand2* geradzahlig sein.

Mit der `SCAN`-Option können Sie nach einem bestimmten Wert in einem Feld suchen.

Der zu suchende Wert (*operand2*) kann entweder als alphanumerische oder Unicode-Konstante (eine in Apostrophen stehende Zeichenkette) oder als Inhalt einer alphanumerischen oder Unicode-Variablen (Datenbankfeld oder Benutzervariable) angegeben werden.

Vorsicht: Dem Wert nachgestellte Leerzeichen in *operand1* werden automatisch eliminiert. Deshalb kann die SCAN-Option nicht zum Suchen nach Leerzeichen verwendet werden. *operand1* und *operand2* dürfen vorangestellte oder eingebettete Leerzeichen enthalten. Falls *operand2* nur aus Leerzeichen besteht, dann gilt, unabhängig vom Wert in *operand1*, die Suche immer als erfolgreich; vergleiche EXAMINE FULL-Statement, wenn vorangestellte Leerzeichen bei der Suche nicht ignoriert werden sollen.

Das Feld, das abgesucht werden soll (*operand1*), kann das Format A, N, P oder U haben. Die SCAN-Operation kann mit den Operatoren EQ (gleich) oder NE (ungleich) angegeben werden.

Die Länge der gesuchten Zeichenkette sollte kürzer als die Länge des abgesuchten Feldes sein. Ist die Länge des Wertes gleich der des Feldes, sollte statt der SCAN-Option ein relationaler Ausdruck mit dem Operator EQUAL TO verwendet werden.

Beispiel für SCAN-Option:

```
** Example 'LOGICX02': SCAN option in logical condition
************************************************************************
DEFINE DATA
LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
*
1 #VALUE   (A4)
1 #COMMENT (A10)
END-DEFINE
*
INPUT 'ENTER SCAN VALUE:' #VALUE
LIMIT 15
*
HISTOGRAM EMPLOY-VIEW FOR NAME
  RESET #COMMENT
  IF NAME = SCAN #VALUE
    MOVE 'MATCH' TO #COMMENT
  END-IF
  DISPLAY NOTITLE NAME *NUMBER #COMMENT
END-HISTOGRAM
*
END
```

Ausgabe des Programms LOGICX02:

```
ENTER SCAN VALUE:
```

Eine Suche nach `LL` führt zu drei Übereinstimmungen bei 15 Namen:

```
        NAME          NMBR      #COMMENT
-------------------- --------- ----------

ABELLAN                        1 MATCH
ACHIESON                       1
ADAM                           1
ADKINSON                       8
AECKERLE                       1
AFANASSIEV                     2
AHL                            1
AKROYD                         1
ALEMAN                         1
ALESTIA                        1
ALEXANDER                      5
ALLEGRE                        1 MATCH
ALLSOP                         1 MATCH
ALTINOK                        1
ALVAREZ                        1
```

SPECIFIED-Option - Prüfen ob ein Wert für einen optionalen Parameter übergeben wird

Syntax:

```
parameter-name [NOT] SPECIFIED
```

Mit dieser Option wird überprüft, ob ein optionaler Parameter in einem aufgerufenen Objekt (Subprogramm, externe Subroutine, Dialog oder ActiveX Control) vom aufrufenden Objekt einen Wert aufgenommen hat oder nicht.

Ein optionaler Parameter ist ein Feld, das mit dem Schlüsselwort `OPTIONAL` im `DEFINE DATA PARAMETER`-Statement des aufgerufenen Objekts definiert worden ist. Wenn ein Feld als `OPTIONAL` definiert wird, kann ein Wert von einem aufrufenden Objekt an dieses Feld übergeben werden.

Im aufrufenden Statement wird die Notation *n*`X` benutzt, um Parameter anzugeben, für die keine Werte übergeben werden.

Wenn Sie einen optionalen Parameter verarbeiten, der keinen Wert empfangen hat, so führt dies zu einem Laufzeit-Fehler. Um solch einen Fehler zu vermeiden, benutzen Sie die

`SPECIFIED`-Option im aufgerufenen Objekt, um zu überprüfen, ob ein optionaler Parameter einen Wert aufgenommen hat oder nicht, und ihn erst dann zu verarbeiten, wenn dies der Fall ist.

parameter-name ist der Name des im `DEFINE DATA PARAMETER`-Statement des aufgerufenen Objekts angegebenen Parameters.

Bei einem nicht als `OPTIONAL` definierten Feld ist die `SPECIFIED`-Bedingung immer `TRUE`.

Beispiel für die SPECIFIED-Option:

Aufrufendes Programm:

```
** Example 'LOGICX07': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #PARM1 (A3)
1 #PARM3 (N2)
END-DEFINE
*
#PARM1 := 'ABC'
#PARM3 := 20
*
CALLNAT 'LOGICX08' #PARM1 1X #PARM3
*
END
```

Aufgerufenes Subprogramm:

```
** Example 'LOGICX08': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA PARAMETER
1 #PARM1 (A3)
1 #PARM2 (N2) OPTIONAL
1 #PARM3 (N2) OPTIONAL
END-DEFINE
*
WRITE '=' #PARM1
*
IF #PARM2 SPECIFIED
  WRITE '#PARM2 is specified'
  WRITE '=' #PARM2
ELSE
  WRITE '#PARM2 is not specified'
* WRITE '=' #PARM2                /* would cause runtime error NAT1322
END-IF
*
IF #PARM3 NOT SPECIFIED
  WRITE '#PARM3 is not specified'
```

```
ELSE
  WRITE '#PARM3 is specified'
  WRITE '=' #PARM3
END-IF
END
```

Ausgabe des Programms `LOGICX07`:

```
Page      1                                                   04-12-15  11:25:41

#PARM1: ABC
#PARM2 is not specified
#PARM3 is specified
#PARM3:  20
```

56 Regeln für arithmetische Operationen

- Initialisierung von Feldern 514
- Kompatibilitätsregeln zur Datenübertragung 514
- Abschneiden und Runden von Feldwerten 517
- Format/Länge von Ergebnisfeldern bei arithmetischen Operationen 517
- Arithmetische Operationen mit Gleitkomma-Zahlen 518
- Arithmetische Operationen mit Datum und Zeit 520
- Formatwahl im Hinblick auf die Verarbeitungszeit 524
- Genauigkeit von Ergebnissen bei arithmetischen Operationen 525
- Fehlerbedingungen bei arithmetischen Operationen 526
- Verarbeitung von Arrays 526

Initialisierung von Feldern

Ein Feld (Datenbankfeld oder Benutzervariable), das in einer arithmetischen Operation als Operand verwendet werden soll, muss mit einem der folgenden Formate definiert werden:

Format	
N	Numerisch ungepackt
P	Numerisch gepackt
I	Integer (Ganzzahl)
F	Floating Point (Gleitkomma)
D	Datum
T	Zeit

Anmerkung: Reporting Mode: Ein Feld, das in einer arithmetischen Operation als Operand verwendet werden soll, muss vorher definiert werden. Benutzervariablen oder Datenbankfelder, die in einer arithmetischen Operation als Ergebnisfeld verwendet werden, müssen nicht vorher definiert werden.

Sobald ein Programm zur Ausführung aufgerufen wird, werden alle im `DEFINE DATA`-Bereich definierten Benutzervariablen und Datenbankfelder mit den entsprechenden Leer- bzw. Nullwerten initialisiert.

Kompatibilitätsregeln zur Datenübertragung

Die Datenübertragung erfolgt mit einem `MOVE`- oder `COMPUTE`-Statement. Die folgende Tabelle fasst die Kompatibilitätsregeln zur Datenübertragung der Formate zusammen, die ein Operand annehmen kann.

Ausgangsfeld	Zielfeld												
	N oder P	A	U	B*n* (*n*<5)	B*n* (*n*>4)	I	L	C	D	T	F	G	O
N oder P	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
A	-	Y	[13]	[1]	[1]	-	-	-	-	-	-	-	-
U	-	[11]	Y	[12]	[12]	-	-	-	-	-	-	-	-
B*n* (*n*<5)	[4]	[2]	[14]	[5]	[5]	Y	-	-	-	Y	Y	-	-
B*n* (*n*>4)	-	[6]	[15]	[5]	[5]	-	-	-	-	-	-	-	-

I	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
L	-	[9]	[16]	-	-	-	Y	-	-	-	-	-	-
C	-	-	-	-	-	-	-	Y	-	-	-	-	-
D	Y	[9]	[16]	Y	-	Y	-	-	Y	[7]	Y	-	-
T	Y	[9]	[16]	Y	-	Y	-	-	[8]	Y	Y	-	-
F	Y	[9] [10]	[10] [16]	[3]	-	Y	-	-	-	Y	Y	-	-
G	-	-	-	-	-	-	-	-	-	-	-	Y	-
O	-	-	-	-	-	-	-	-	-	-	-	-	Y

Legende:

Y	Übertragung möglich.
-	Übertragung nicht möglich.
[ ]	Übertragung möglich. Die Ziffern in eckigen Klammern [ ] beziehen sich auf die entsprechende Regel für die Übertragung (siehe unten).

Umsetzung von Daten in ein anderes Format

Bei der Umsetzung von Werten in ein anderes Format gelten folgende Regeln:

1. **Von Alphanumerisch (A) in Binär (B):**
   Der Wert wird Byte für Byte von links nach rechts übertragen. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird der Wert entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.

2. **Von Numerisch (N), Gepackt (P), Ganzzahl (I) und Binär (B) mit 1-4 Bytes Länge in Alphanumerisch:**
   Der Wert wird in ungepacktes Format umgesetzt und linksbündig in das Zielfeld übertragen, wobei vorangestellte Nullen weggelassen werden und der Rest des Feldes mit Leerzeichen aufgefüllt wird. Bei negativen numerischen Werten wird das Vorzeichen in die Hexadezimalnotation `Dx` umgesetzt. Ein Komma (Dezimalpunkt) im Ausgangswert wird nicht berücksichtigt, und alle Stellen vor und nach dem Komma werden als ganze Zahl interpretiert.

3. **Von Numerisch (N), Gepackt (P), Ganzzahl (I), Gleitkomma (F) in Binär (B) mit 1-4 Bytes Länge:**
   Der Wert wird in binäres Format umgesetzt (4 Bytes). Ein Komma (Dezimalpunkt) wird ignoriert, die Stellen vor und nach dem Komma werden als ganze Zahl behandelt. Je nach Vorzeichen ist die Binärzahl entweder positiv oder das Zweierkomplement des Wertes.

4. **Von Binär (B) mit 1-4 Bytes Länge in Numerisch (N):**
   Der Wert wird umgesetzt und rechtsbündig übertragen, der Rest des Feldes wird mit Nullen aufgefüllt. Binäre Werte von 1 bis 3 Bytes Länge werden immer als positiv interpretiert. Bei 4 Byte langen binären Werten bestimmt das erste (linke) Bit das Vorzeichen: 1 = negativ, 0 =

positiv. Ein Komma (Dezimalpunkt) im Zielfeld wird nicht berücksichtigt, und alle Stellen vor und nach dem Komma werden als ganze Zahl interpretiert.

5. **Von Binär (B) in Binär (B):**
   Der Wert wird Byte für Byte von rechts nach links übertragen, und der Rest des Feldes wird mit Nullen aufgefüllt.
6. **Von Binär (B) mit mehr als 4 Bytes in Alphanumerisch (A):**
   Der Wert wird Byte für Byte von links nach rechts übertragen. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird der Wert entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.
7. **Von Datum (D) in Zeit (T):**
   Das Datum wird in Zeit umgesetzt, und zwar ausgehend von der Zeit `00:00:00:0`.
8. **Von Zeit (T) in Datum (D):**
   Die Zeitkomponente wird abgeschnitten und nur die Datumskomponente des Zeitfeldes wird in das Datumsfeld übertragen.
9. **Von Logisch (L), Datum (D), Zeit (T), Gleitkomma (F) in Alphanumerisch (A):**
   Der Wert wird in Anzeigeform umgesetzt und linksbündig übertragen.
10. **Gleitkomma (F):**
    Wird ein Gleitkomma-Wert in ein alphanumerisches oder Unicode-Feld übertragen, das zu kurz ist, wird die Mantisse entsprechend gekürzt.
11. **Von Unicode (U) in Alphanumerisch (A):**
    Der Unicode-Wert wird anhand der Library ICU (International Components for Unicode) entsprechend der voreingestellten (Default-)Codepage (Wert der Systemvariablen `*CODEPAGE`) in alphanumerische Zeichencodes umgesetzt. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.
12. **Von Unicode (U) in Binär (B):**
    Der Wert wird Code Unit für Code Unit von links nach rechts verschoben. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt. Die Länge des binären Zielfeldes muss geradezahlig sein.
13. **Von Alphanumerisch (A) in Unicode (U):**
    Der alphanumerische Wert wird unter Benutzung der Library ICU (International Components for Unicode) von der voreingestellten (Default-)Codepage in einen Unicode-Wert umgesetzt. Je nach Länge des Zielfeldes und der Anzahl der Code Units wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.
14. **Von N, P, I und Binär (Länge 1–4) in Unicode (U):**
    Der Wert wird in ungepacktes Format konvertiert, aus dem dann ein alphanumerischer Wert durch die Unterdrückung von führenden Nullen erhalten werden kann. Bei negativen numerischen Werten wird das Vorzeichen in die hexadezimale Notation `Dx` umgesetzt. Ein Dezimalpunkt im numerischen Wert wird ignoriert. Alle Ziffern vor und nach dem Dezimalpunkt werden als ein Ganzzahlwert (Integer) behandelt. Der Ergebniswert wird von alphanumerisch

in Unicode umgesetzt. Je nach Länge des Zielfeldes und der Anzahl der Code Units wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.

15. **Von Binär (B) mit mehr als 4 Bytes in Unicode (U):**
Der Wert wird Byte für Byte von links nach rechts verschoben. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt. Die Länge des binären Ausgangsfeldes muss geradzahlig sein.

16. **Von L, D, T, F in U:**
Die Werte werden in ein alphanumerisches Anzeigeformat konvertiert. Der Ergebniswert wird von alphanumerisch in Unicode umgesetzt und linksbündig ausgerichtet.

Wenn das Ausgangs- und Zielformat identisch sind, kann je nach der Länge und der Anzahl der Bytes (Format A und B) oder Code-Einheiten (Format U) das Ergebnis abgeschnitten oder mit Leerzeichen (Format A und U) oder führenden binären Nullen (Format B) aufgefüllt werden.

Siehe auch *Dynamische Variablen benutzen*.

Abschneiden und Runden von Feldwerten

Die folgenden Regeln gelten für das Abschneiden und Runden von Feldwerten:

- Numerische Felder: vorangestellte Stellen dürfen nur abgeschnitten werden, falls ihr Wert Null ist. Stellen nach einem ausdrücklich angegebenen oder implizierten Komma (Dezimalpunkt) dürfen abgeschnitten werden.
- Alphanumerische Felder: Nachfolgende Stellen dürfen abgeschnitten werden.
- Bei Verwendung der Option `ROUNDED` wird die letzte Stelle im Feld aufgerundet, falls die erste abgeschnittene Stelle größer/gleich 5 ist. Zur Ergebnisgenauigkeit einer Division siehe auch Abschnitt *Genauigkeit von Ergebnissen bei arithmetischen Operationen*.

Format/Länge von Ergebnisfeldern bei arithmetischen Operationen

Die folgende Tabelle zeigt Format/Länge von Ergebnisfeldern bei arithmetischen Operationen:

	I1	I2	I4	N oder P	F4	F8
I1	I1	I2	I4	P*	F4	F8
I2	I2	I2	I4	P*	F4	F8
I4	I4	I4	I4	P*	F4	F8
N oder P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8

	I1	I2	I4	N oder P	F4	F8
F8	F8	F8	F8	F8	F8	F8

Auf Großrechnern wird Format/Länge F8 anstatt F4 für eine verbesserte Ergebnisgenauigkeit einer arithmetischen Operation benutzt.

P* ergibt sich aus der ganzzahligen Länge und Genauigkeit der einzelnen Operanden je nach Operation (siehe Abschnitt *Genauigkeit von Ergebnissen bei arithmetischen Operationen*.

Für Format I gelten die folgenden dezimalen Ganzzahl-Längen und möglichen Werte:

Format/Länge	Dezimale Ganzzahl-Länge	Mögliche Werte
I1	3	-128 bis 127
I2	5	-32768 bis 32767
I4	10	-2147483648 bis 2147483647

Arithmetische Operationen mit Gleitkomma-Zahlen

Folgende Themen werden behandelt:

- Einige allgemeine Hinweise
- Genauigkeit von Gleitkomma-Zahlen
- Konvertierung in Gleitkomma-Darstellung
- Plattform-abhängige Unterschiede

Einige allgemeine Hinweise

Gleitkomma-Zahlen (Format F) werden ebenso wie Ganzzahlen (Format I) als Summe von Zweierpotenzen dargestellt, wohingegen ungepackte und gepackte Zahlen (Formate N und P) als Summe von Zehnerpotenzen dargestellt werden.

Bei ungepackten oder gepackten Zahlen ist die Position des Dezimalkommas fest. Bei Gleitkomma-Zahlen dagegen ist (wie der Name schon andeutet) die Position des Dezimalkommas „gleitend“, d.h. seine Position ist nicht fest, sondern hängt vom tatsächlichen Wert der Zahl ab.

Gleitkomma-Zahlen sind unverzichtbar bei der Berechnung trigonometrischer und mathematischer Funktionen wie etwa Sinus oder Logarithmus.

Genauigkeit von Gleitkomma-Zahlen

Die Genauigkeit von Gleitkomma-Zahlen an sich ist begrenzt:

- Bei einer Variablen mit Format/Länge F4 ist die Genauigkeit auf etwa 7 Stellen begrenzt.
- Bei einer Variablen mit Format/Länge F8 ist die Genauigkeit auf 15 Stellen begrenzt.

Werte mit einer größeren Anzahl signifikanter Stellen lassen sich nicht exakt als Gleitkomma-Zahlen darstellen. Unabhängig von der Zahl zusätzlicher Vor- oder Nachkommastellen kann eine Gleitkomma-Zahl nur die ersten 7 bzw. 15 Stellen abdecken.

Eine Ganzzahl lässt sich nur exakt in einer Variablen mit Format/Länge F4 darstellen, wenn ihr absoluter Wert nicht größer als $2^{23}-1$ ist.

Konvertierung in Gleitkomma-Darstellung

Wenn ein alphanumerischer, ungepackter numerischer oder gepackter numerischer Wert in Gleitkomma-Format umgesetzt wird (zum Beispiel bei einer Zuweisung), muss auch die Darstellungsform geändert werden, d.h. eine Summe von Zehnerpotenzen muss in eine Summe von Zweierpotenzen konvertiert werden.

Folglich lassen sich nur Zahlen, die als endliche Summe von Zweierpotenzen darstellbar sind, exakt darstellen; alle anderen Zahlen lassen sich nur näherungsweise darstellen.

Beispiele:

Diese Zahl hat eine exakte Gleitkomma-Darstellung:

$$1.25 = 2^{0} + 2^{-2}$$

Diese Zahl ist eine periodische Gleitkomma-Zahl ohne exakte Darstellung:

$$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \dots$$

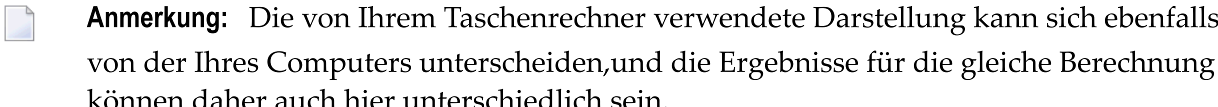
Daher kann die Konvertierung von alphanumerischen, ungepackt numerischen oder gepackt numerischen Werten in Gleitkomma-Werte, und umgekehrt, zu kleineren Fehlern führen.

Plattform-abhängige Unterschiede

Aufgrund der unterschiedlichen Hardware-Architektur ist die Darstellung von Gleitkomma-Zahlen auf Großrechnern anders als auf anderen Plattformen. Dies erklärt, warum dieselbe Anwendung bei Gleitkomma-Berechnungen auf verschiedenen Plattformen andere Ergebnisse liefert.

Der mögliche Wertebereich auf einem Großrechner ist (ca.):

- $\pm 1.17 * 10^{-38}$ to $\pm 3.40 * 10^{38}$ für F4-Variablen,
- $\pm 2.22 * 10^{-308}$ to $\pm 1.79 * 10^{308}$ für F8-Variablen.



Anmerkung: Die von Ihrem Taschenrechner verwendete Darstellung kann sich ebenfalls von der Ihres Computers unterscheiden,und die Ergebnisse für die gleiche Berechnung können daher auch hier unterschiedlich sein.

Arithmetische Operationen mit Datum und Zeit

Mit Feldern der Formate D (Datum) und T (Time = Zeit) sind nur Addition, Subtraktion, Multiplikation und Division erlaubt. Multiplikation und Division sind nur bei Zwischenergebnissen von Addition und Subtraktion zulässig.

Datums-/Zeitwerte können addiert bzw. voneinander subtrahiert werden; oder Ganzzahl-Werte (ohne Nachkommastellen) können zu/von Datums-/Zeitwerten addiert/subtrahiert werden. Solche ganzzahligen Werte können in Feldern der Formate N, P, I, D oder T enthalten sein.

Die Zwischenergebnisse einer solchen Addition oder Subtraktion können als Multiplikand oder Dividend in einer nachfolgenden Operation verwendet werden.

Von ganzzahligen Werten, die zu einem Datumswert addiert oder von einem Datumswert subtrahiert werden, wird angenommen, dass es sich um Tage handelt. Von ganzzahligen Werten, die zu einem Zeitwert addiert oder von einem Zeitwert subtrahiert werden, wird angenommen, dass es sich um Zehntelsekunden handelt.

Bei arithmetischen Operationen mit Datum und Zeit gelten gewisse Einschränkungen, und zwar aufgrund von Naturals interner Behandlung von Datums- und Zeitarithmetik, wie im folgenden erläutert.

Intern behandelt Natural eine arithmetische Operation mit Datums- bzw. Zeitvariablen wie folgt:

```
COMPUTE result-field = operand1 +/- operand2
```

Das obige Statement wird aufgelöst als:

1. *intermediate-result = operand1 +/- operand2*
2. *result-field = intermediate-result*

Das heißt, zunächst berechnet Natural das Ergebnis der Addition/Subtraktion, und erst danach weist es das Ergebnis dem Ergebnisfeld zu.

Komplexere arithmetische Operationen werden nach dem gleichen Muster aufgelöst:

```
COMPUTE result-field = operand1 +/- operand2 +/- operand3 +/- operand4
```

Das obige Statement wird aufgelöst als:

1. *intermediate-result1 = operand1 +/- operand2*
2. *intermediate-result2 = intermediate-result1 +/- operand3*
3. *intermediate-result3 = intermediate-result2 +/- operand4*
4. *result-field = intermediate-result3*

Die Auflösung bei der Multiplikation und Division ist ähnlich wie die Auflösung bei der Addition und Subtraktion.

Das interne Format eines solchen Zwischenergebnisses (*intermediate-result*) hängt vom Format der einzelnen Operanden ab, wie die folgenden Tabellen zeigen.

Addition

Die folgende Tabelle zeigt das Format vom Zwischenergebnis einer Addition (*intermediate-result* = *operand1* + *operand2*):

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
D	D	Di
D	T	T
D	Di, Ti, N, P, I	D
T	D, T, Di, Ti, N, P, I	T
Di, Ti, N, P, I	D	D
Di, Ti, N, P, I	T	T
Di, N, P, I	Di	Di

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
Ti, N, P, I	Ti	Ti
Di	Ti, N, P, I	Di
Ti	Di, N, P, I	Ti

Subtraktion

Die folgende Tabelle zeigt das Format des Zwischenergebnisses einer Subtraktion (*intermediate-result* = *operand1* - *operand2*):

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
D	D	Di
D	T	Ti
D	Di, Ti, N, P, I	D
T	D, T	Ti
T	Di, Ti, N, P, I	T
Di, N, P, I	D	Di
Di, N, P, I	T	Ti
Di	Di, Ti, N, P, I	Di
Ti	D, T, Di, Ti, N, P, I	Ti
N, P, I	Di, Ti	P12

Multiplikation oder Division

Die folgende Tabelle zeigt das Format des Zwischenergebnisses einer Multiplikation: (*intermediate-result* = *operand1* * *operand2*) oder Division (*intermediate-result* = *operand1* / *operand2*):

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
D	D, Di, Ti, N, P, I	Di
D	T	Ti
T	D, T, Di, Ti, N, P, I	Ti
Di	T	Ti
Di	D, Di, Ti, N, P, I	Di
Ti	D	Di
Ti	Di, T, Ti, N, P, I	Ti
N, P, I	D, Di	Di
N, P, I	T, Ti	Ti

Interne Zuweisungen

Di ist ein Wert im internen Datumsformat; Ti ist ein Wert im internen Zeitformat; solche Werte können zwar in weiteren arithmetischen Datums-/Zeitoperationen verwendet werden, aber sie können keinem Ergebnisfeld vom Format D zugewiesen werden (siehe Zuweisungstabelle unten).

Bei komplexen arithmetischen Operationen, bei denen ein Zwischenergebnis im internen Format Di bzw. Ti als Operand für eine weitere Addition/Subtraktion/Multiplikation/Division verwendet wird, wird davon ausgegangen, dass es Format D bzw. T hat.

Die folgende Tabelle zeigt, welche Zwischenergebnisse intern welchen Ergebnisfeldern zugewiesen werden können (`result-field` = `intermediate-result`).

Format von `result-field`	Format von `intermediate-result`	Zuweisung möglich
D	D, T	ja
D	Di, Ti, N, P, I	nein
T	D, T, Di, Ti, N, P, I	ja
N, P, I	D, T, Di, Ti, N, P, I	ja

Ein Ergebnisfeld vom Format D oder T darf keinen negativen Wert enthalten.

Beispiele 1 und 2 (ungültig):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D)
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

Diese Operationen sind nicht möglich, da das Zwischenergebnis der Addition bzw. Subtraktion Format Di hätte, und ein Wert vom Format D*i* keinem Ergebnisfeld vom Format D zugewiesen werden kann.

Beispiele 3 und 4 (ungültig):

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T)
COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

Diese Operationen sind nicht möglich, da das Zwischenergebnis der Addition bzw. Subtraktion Format T*i* hätte, und ein Wert vom Format T*i* keinem Ergebnisfeld vom Format D zugewiesen werden kann.

Beispiel 5 (gültig):

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

Diese Operation ist möglich. Zunächst wird `DATE3` von `DATE2` subtrahiert, woraus sich ein Zwischenergebnis vom Format D *i* ergibt; dann wird dieses Zwischenergebnis zu `TIME3` hinzuaddiert, woraus sich ein Zwischenergebnis vom Format T ergibt; und schließlich wird dieses zweite Zwischenergebnis dem Ergebnisfeld `DATE1` zugewiesen.

Beispiele 6 und 7 (ungültig):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) * 2
COMPUTE TIME1 (T) = TIME2 (T) - TIME3 (T) / 3
```

Diese Operationen sind nicht möglich, da die versuchte Multiplikation bzw. Division mit Datums-/Zeitfeldern und nicht mit Zwischenergebnissen durchgeführt wird.

Beispiel 8 (gültig):

```
COMPUTE DATE1 (D) = DATE2 (D) + (DATE3(D) - DATE4 (D)) * 2
```

Diese Operation ist möglich. Zunächst wird `DATE4` von `DATE3` subtrahiert, woraus sich ein Zwischenergebnis vom Format D *i* ergibt; dann wird dieses Zwischenergebnis mit 2 multipliziert, woraus sich ein Zwischenergebnis vom Format D *i* ergibt; dieses Zwischenergebnis wird zu `DATE2` addiert, woraus sich ein Zwischenergebnis vom Format D ergibt; und schließlich wird dieses dritte Zwischenergebnis dem Ergebnisfeld `DATE1` zugewiesen.

Wenn Sie einen Format-T-Wert einem Format-D-Feld zuweisen, müssen Sie dafür sorgen, dass der Zeitwert eine gültige Datumskomponente enthält.

Formatwahl im Hinblick auf die Verarbeitungszeit

Bei arithmetischen Operationen hat die Wahl der richtigen Feldformate starken Einfluss auf die Verarbeitungszeit:

Bei kaufmännischen Berechnungen empfiehlt es sich, nur Felder mit dem Format I (Integer) zu verwenden.

Bei wissenschaftlichen Berechnungen empfiehlt es sich, nur Felder mit dem Format F (Gleitkomma-Format) zu verwenden.

Werden die numerischen Formate N und P mit dem Format F vermischt, erfolgt intern eine Umsetzung in Format F; diese Umsetzung führt zu einer nicht unbeträchtlichen CPU-Beanspru-

chung. Daher sollte es möglichst vermieden werden, bei arithmetischen Operationen unterschiedliche Formate zu verwenden.

Genauigkeit von Ergebnissen bei arithmetischen Operationen

Operation	Stellen vor dem Komma	Stellen nach dem Komma
Addition/Subtraktion	Fi + 1 oder Si + 1 (das jeweils größere)	Fd oder Sd (das jeweils größere)
Multiplikation	Fi + Si + 2	Fd + Sd (höchstens 7)
Division	Fi + Sd	(siehe unten)
Potenzierung	15 - Fd (Siehe *Ausnahme* unten)	Fd
Quadratwurzel	Fi	Fd

Dabei ist:

F	Erster Operand
S	Zweiter Operand
R	Ergebnis
i	Stellen vor dem Komma (Dezimalpunkt)
d	Stellen nach dem Komma (Dezimalpunkt)

Ausnahme:

Wenn die Hochzahl eine oder mehrere Stellen hinter dem Komma (Dezimalpunkt) aufweist, wird die Potenzierung intern im Gleitkomma-Format ausgeführt und das Ergebnis hat ebenfalls Gleitkomma-Format. Weitere Informationen siehe Abschnitt *Arithmetische Operationen mit Gleitkomma-Zahlen*.

Nachkommastellen bei Divisionsergebnissen

Die Genauigkeit des Ergebnisses einer Division hängt davon ab, ob ein Ergebnisfeld vorhanden ist oder nicht:

- Ist ein Ergebnisfeld vorhanden, ist die Genauigkeit: `Rd` oder `Fd` (das jeweils größere) *.
- Ist kein Ergebnisfeld vorhanden, ist die Genauigkeit: `Fd` oder `Sd` (das jeweils größere) *.

* Bei Verwendung der `ROUNDED`-Option erhöht sich die Ergebnisgenauigkeit intern um eine Stelle, bevor das Ergebnis tatsächlich gerundet wird.

Ein Ergebnisfeld ist vorhanden (bzw. wird als vorhanden angenommen) in einem `COMPUTE`- und `DIVIDE`-Statement sowie in einer logischen Bedingung, in der die Division hinter dem Vergleichsoperator steht (z.B.: `IF #A = #B / #C THEN ...`).

Ein Ergebnisfeld ist nicht vorhanden (bzw. wird als nicht vorhanden angenommen) in einer logischen Bedingung, in der die Division vor dem Vergleichsoperator steht (z.B.: IF #B / #C = #A THEN ...).

Ausnahme:

Wenn Dividend und Divisor Ganzzahlen sind und mindestens eine davon eine Variable ist, dann ist auch das Divisionsergebnis eine Ganzzahl (unabhängig von der Genauigkeit des Ergebnisfeldes sowie der Verwendung der ROUNDED-Option).

Genauigkeit von Ergebnissen bei arithmetischen Ausdrücken

Die Genauigkeit von arithmetischen Ausdrücken, zum Beispiel #A / (#B * #C) + #D * (#E - #F + #G), wird von der Auswertung der Ergebnisse von arithmetischen Operationen in ihrer Verarbeitungsreihenfolge hergeleitet. Weitere Informationen zu arithmetischen Ausdrücken siehe *arithmetic-expression* in der Beschreibung des COMPUTE-Statements.

Fehlerbedingungen bei arithmetischen Operationen

Bei einer Addition, Subtraktion, Multiplikation oder Division erhalten Sie einen Fehler, wenn das Ergebnis (insgesamt, d.h. vor und nach dem Komma) mehr als 31 Stellen hat.

Bei einer Potenzierung erhalten Sie unter einer der folgenden Bedingungen einen Fehler:

- wenn die Basis gepacktes Format hat und entweder das Ergebnis mehr als 16 Stellen oder ein Zwischenergebnis mehr als 15 Stellen hat;
- wenn die Basis Gleitkomma-Format hat und das Ergebnis größer ist als ca. $7 * 10^{75}$.

Verarbeitung von Arrays

Grundsätzlich gelten folgende Regeln:

- Alle Skalar-Operationen können auf Array-Elemente angewandt werden, die aus einer einzigen Ausprägung bestehen.
- Wenn eine Variable mit einem konstanten Wert definiert ist (z.B. #FIELD (I2) CONSTANT <8>), dann wird der Wert der Variablen bei der Kompilierung zugewiesen, und die Variable wird als Konstante behandelt. Falls eine solche Variable in einem Array-Index verwendet wird, bedeutet dies, dass die betreffende Dimension eine *bestimmte* Anzahl von Ausprägungen hat.
- Bei einer Zuweisung bzw. einem Vergleich zwischen zwei Arrays mit unterschiedlich vielen Dimensionen wird angenommen, dass die „fehlende“ Dimension in dem Array mit weniger Dimensionen (1:1) ist.

Beispiel: Wenn das Array `#ARRAY1 (1:2)` dem Array `#ARRAY2 (1:2,1:2)` zugewiesen wird, wird für `#ARRAY1` angenommen, dass es `#ARRAY1 (1:1,1:2)` ist.

Folgende Themen werden behandelt:

- Definition von Array-Dimensionen
- Zuweisungen bei Arrays
- Vergleiche mit Arrays
- Arithmetische Operationen mit Arrays

Definition von Array-Dimensionen

Die erste, zweite und dritte Dimension eines Arrays werden wie folgt definiert:

Dimensionen	Eigenschaften
3	#a3 (3. Dim., 2. Dim., 1. Dim.)
2	#a2 (2. Dim., 1. Dim.)
1	#a1 (1. Dim.)

Zuweisungen bei Arrays

Wenn Sie einen Array-Bereich einem anderen Array-Bereich zuweisen, erfolgt die Zuweisung Element für Element.

Beispiel:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE
*
MOVE #ARRAY(2:4) TO #ARRAY(3:5)
/* is identical to
/* MOVE #ARRAY(2) TO #ARRAY(3)
/* MOVE #ARRAY(3) TO #ARRAY(4)
/* MOVE #ARRAY(4) TO #ARRAY(5)
/*
/* #ARRAY contains  10,20,20,20,20
```

Wenn Sie eine einzelne Ausprägung einem Array-Bereich zuweisen, wird jedes Element des Bereiches mit dem Wert der einzelnen Ausprägung gefüllt. (Bei einer mathematischen Funktion wird jedes Element des Bereiches mit dem Ergebnis der Funktion gefüllt.)

Bevor eine Zuweisung ausgeführt wird, werden die einzelnen Dimensionen der betroffenen Arrays miteinander verglichen, um zu prüfen, ob sie eine der unten aufgeführten Bedingungen erfüllen.

Die Dimensionen werden dabei unabhängig voneinander verglichen; d.h. die 1. Dimension des einen Arrays wird mit der 1. Dimension des anderen Arrays verglichen, die 2. Dimension des einen Arrays wird mit der 2. Dimension des anderen Arrays verglichen, und die 3. Dimension des einen Arrays wird mit der 3. Dimension des anderen Arrays verglichen.

Die Zuweisung von Werten eines Arrays an ein anderes Array ist nur unter einer der folgenden Bedingungen erlaubt:

- Die zwei verglichenen Dimensionen haben die gleiche Anzahl von Ausprägungen.
- Die zwei verglichenen Dimensionen haben beide eine unbestimmte Anzahl von Ausprägungen.
- Die Dimension, die einer anderen Dimension zugewiesen wird, besteht aus einer einzelnen Ausprägung.

Beispiel für Array-Zuweisungen:

Das folgende Programm zeigt, welche Zuweisungen zwischen Arrays möglich sind.

```
 DEFINE DATA LOCAL
 1 A1   (N1/1:8)
 1 B1   (N1/1:8)
 1 A2   (N1/1:8,1:8)
 1 B2   (N1/1:8,1:8)
 1 A3   (N1/1:8,1:8,1:8)
 1 I    (I2)          INIT <4>
 1 J    (I2)          INIT <8>
 1 K    (I2)          CONST <8>
 END-DEFINE
 *
 COMPUTE A1(1:3) = B1(6:8)                                      /* allowed
 COMPUTE A1(1:I) = B1(1:I)                                      /* allowed
 COMPUTE A1(*)   = B1(1:8)                                      /* allowed
 COMPUTE A1(2:3) = B1(I:I+1)                                    /* allowed
 COMPUTE A1(1)   = B1(I)                                        /* allowed
 COMPUTE A1(1:I) = B1(3)                                        /* allowed
 COMPUTE A1(I:J) = B1(I+2)                                      /* allowed
 COMPUTE A1(1:I) = B1(5:J)                                      /* allowed
 COMPUTE A1(1:I) = B1(2)                                        /* allowed
 COMPUTE A1(1:2) = B1(1:J)                                      /* NOT ALLOWED
(NAT0631)
 COMPUTE A1(*)   = B1(1:J)                                      /* NOT ALLOWED
(NAT0631)
 COMPUTE A1(*)   = B1(1:K)                                      /* allowed
 COMPUTE A1(1:J) = B1(1:K)                                      /* NOT ALLOWED
(NAT0631)
 *
 COMPUTE A1(*)       = B2(1,*)                                  /* allowed
 COMPUTE A1(1:3)     = B2(1,I:I+2)                              /* allowed
 COMPUTE A1(1:3)     = B2(1:3,1)                                /* NOT ALLOWED
```

```
(NAT0631)
 *
 COMPUTE A2(1,1:3)   = B1(6:8)                                   /* allowed
 COMPUTE A2(*,1:I)   = B1(5:J)                                   /* allowed
 COMPUTE A2(*,1)     = B1(*)                                     /* NOT ALLOWED
(NAT0631)
 COMPUTE A2(1:I,1)   = B1(1:J)                                   /* NOT ALLOWED
(NAT0631)
 COMPUTE A2(1:I,1:J) = B1(1:J)                                   /* allowed
 *
 COMPUTE A2(1,I)     = B2(1,1)                                   /* allowed
 COMPUTE A2(1:I,1)   = B2(1:I,2)                                 /* allowed
 COMPUTE A2(1:2,1:8) = B2(I:I+1,*)                               /* allowed
 *
 COMPUTE A3(1,1,1:I)   = B1(1)                                   /* allowed
 COMPUTE A3(1,1,1:J)   = B1(*)                                   /* NOT ALLOWED
(NAT0631)
  COMPUTE A3(1,1,1:I)   = B1(1:I)                                /* allowed
 COMPUTE A3(1,1:2,1:I) = B2(1,1:I)                               /* allowed
 COMPUTE A3(1,1,1:I)   = B2(1:2,1:I)                             /* NOT ALLOWED
(NAT0631)
 END
```

Vergleiche mit Arrays

Grundsätzlich gilt Folgendes: Wenn mehrdimensionale Arrays miteinander verglichen werden, werden die einzelnen Dimensionen unabhängig voneinander behandelt; d.h. die 1. Dimension des einen Arrays wird mit der 1. Dimension des anderen Arrays verglichen, die 2. Dimension des einen Arrays wird mit der 2. Dimension des anderen Arrays verglichen, und die 3. Dimension des einen Arrays wird mit der 3. Dimension des anderen Arrays verglichen.

Der Vergleich zweier Array-Dimensionen ist nur unter einer der folgenden Bedingungen erlaubt:

- Die zwei verglichenen Dimensionen haben die gleiche Anzahl von Ausprägungen.
- Die zwei verglichenen Dimensionen haben beide eine unbestimmte Anzahl von Ausprägungen.
- Alle Dimensionen des einen Arrays bestehen jeweils aus einer einzelnen Ausprägung.

Beispiel für Array-Vergleiche:

Das folgende Programm zeigt, welche Vergleiche zwischen Arrays möglich sind.

```
DEFINE DATA LOCAL
1 A3  (N1/1:8,1:8,1:8)
1 A2  (N1/1:8,1:8)

1 A1  (N1/1:8)
1 I   (I2)   INIT <4>
1 J   (I2)   INIT <8>
1 K   (I2)   CONST <8>
END-DEFINE
*
IF A2(1,1)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(*)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I -3:I+4)       THEN IGNORE END-IF /* allowed
IF A2(1,5:J)   = A1(1:I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1:I)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,*)     = A1(1:K)            THEN IGNORE END-IF /* allowed
*
IF A2(1,1)     = A2(1,1)            THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A2(1,I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(1,1:8)          THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(I,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I+1:J)        THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I:I+1)        THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(*,1)     = A2(1,*)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,1:I)   = A1(2,1:K)          THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
*
IF A3(1,1,*)   = A2(1,*)            THEN IGNORE END-IF /* allowed
IF A3(1,1,*)   = A2(1,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,1:I+1)        THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,I:J)          THEN IGNORE END-IF /* allowed
END
```

Wenn Sie zwei Array-Bereiche miteinander vergleichen, beachten Sie bitte, dass die folgenden zwei Ausdrücke zu unterschiedlichen Ergebnissen führen:

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
NOT #ARRAY1(*) = #ARRAY2(*)
```

Beispiel:

- **Bedingung A:**

  ```
  IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
  ```

  Dies entspricht:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Bedingung A ist also erfüllt, wenn die erste Ausprägung von `#ARRAY1` ungleich der ersten Ausprägung von `#ARRAY2` ist und die zweite Ausprägung von `#ARRAY1` ungleich der zweiten Ausprägung von `#ARRAY2` ist.

- **Bedingung B:**

  ```
  IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
  ```

  Dies entspricht:

  ```
  IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
  ```

  Dies wiederum entspricht:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Bedingung B ist also erfüllt, wenn *entweder* die erste Ausprägung von `#ARRAY1` ungleich der ersten Ausprägung von `#ARRAY2` ist *oder* die zweite Ausprägung von `#ARRAY1` ungleich der zweiten Ausprägung von `#ARRAY2` ist.

Arithmetische Operationen mit Arrays

Eine allgemeine Regel zu Arrays lautet, dass die Anzahl der Ausprägungen der entsprechenden Dimensionen gleich sein muss.

Das folgende Beispiel veranschaulicht diese Regel:

```
#c(2:3,2:4) := #a(3:4,1:3) + #b(3:5)
```

Mit anderen Worten:

Array	Dimension	Anzahl der Ausprägungen	Bereich
#c	2.	2	2:3
#c	1.	3	2:4
#a	2.	2	3:4
#a	1.	3	1:3
#b	1.	3	3:5

Die Operation wird Element für Element durchgeführt

Anmerkung: Eine arithmetische Operation einer unterschiedlichen Anzahl von Dimensionen ist zulässig.

Für das obige Beispiel werden die folgenden Operationen ausgeführt:

```
#c(2,2) := #a(3,1) + #b(3)

#c(2,3) := #a(3,2) + #b(4)

#c(2,4) := #a(3,3) + #b(5)

#c(3,2) := #a(4,1) + #b(3)

#c(3,3) := #a(4,2) + #b(4)

#c(3,4) := #a(4,3) + #b(5)
```

In arithmetischen Operationen (in `COMPUTE`-, `ADD`- oder `MULTIPLY`-Statements) können Array-Bereiche auf folgende Arten verwendet werden. In den Beispielen 1 - 4 muss die Anzahl der Ausprägungen der entsprechenden Dimensionen gleich sein.

1. *range + range = range*.

   Die Addition wird Element für Element ausgeführt.

2. `range * range = range`.

   Die Multiplikation wird Element für Element ausgeführt.

3. `scalar + range = range`.

   Der Skalarwert wird zu jedem Element des Bereichs addiert.

4. `range * scalar = range`.

   Jedes Element des Bereichs wird mit dem Skalarwert multipliziert.

5. `range + scalar = scalar`.

   Jedes Element des Bereichs wird zum Skalarwert addiert und das Ergebnis im Skalar ausgegeben.

6. `scalar * range = scalar2`.

   Der Skalarwert wird mit jedem Element des Arrays multipliziert und das Ergebnis in `scalar2` ausgegeben.

Weil, wie aus den obigen Beispielen hervorgeht, bei arithmetischen Operationen Zwischenergebnisse erzeugt werden, wird das Ergebnis von sich überlappenden Indexbereichen Element für Element in einem Zwischenergebnis-Array berechnet, und schließlich wird das Zwischenergebnis-Array dem Ergebnisfeld zugewiesen.

Beispiel:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE

#ARRAY(3:5) := #ARRAY(2:4) + 1

/* A temporary array for the
/* intermediate result values is
/* generated implicitly: #temp(1:3).
/* The following operations are
/* performed internally:
/* #temp(1) := #ARRAY(2) + 1
/* #temp(2) := #ARRAY(3) + 1
/* #temp(3) := #ARRAY(4) + 1
/* #ARRAY(3:5) := #temp(1:3)
/*
/* #ARRAY contains  10,20,21,31,41
```

57 Natural-Subprogramme aus 3GL-Programmen aufrufen

- Parameter vom 3GL-Programm an das Subprogramm übergeben 536
- Beispiel: Aufruf eines Natural-Subprogramms von einem 3GL-Programm 537

Natural-Subprogramme können von einem Programmierobjekt aufgerufen werden, das in einer Programmiersprache der dritten Generation (3GL) geschrieben ist. Das aufrufende Programm kann in einer beliebigen Programmiersprache geschrieben sein, die ein Standard-`CALL`-Interface unterstützt.

Aus diesem Grund stellt Natural das Interface `ncxr_callnat` zur Verfügung. Das 3GL-Programm ruft dieses Interface auf, indem es den Namen des gewünschten Subprogramms spezifiziert.

Anmerkung: Natural muss vorher aktiviert worden sein; d.h. das aufrufende 3GL-Programm muss wiederum von einem Natural-Objekt mit einem `CALL`-Statements aufgerufen worden sein.

Das Subprogramm wird so ausgeführt als wäre es von einem anderen Natural-Objekt mit einem `CALLNAT`-Statements aufgerufen worden.

Wenn die Verarbeitung des Subprogrammes stoppt (entweder mit dem `END`-Statement oder mit einem `ESCAPE ROUTINE`-Statement), wird die Kontrolle an das 3GL-Programm zurückgegeben.

Parameter vom 3GL-Programm an das Subprogramm übergeben

Parameter können vom aufrufenden 3GL-Programm an das Natural-Subprogramm übergeben werden. Für die Übergabe der Parameter gelten dieselben Regeln wie für die Übergabe mit einem `CALL`-Statement.

Das 3GL-Programm ruft das Natural-Interface `ncxr_callnat` mit vier Parametern auf:

- Der erste Parameter ist der Name des Natural-Subprogramms, das aufgerufen werden soll.
- Der zweite Parameter enthält die Anzahl der Parameter, die an das Subprogramm übergeben werden sollen.
- Der dritte Parameter enthält die Adresse der Tabelle, die die Adressen der Parameter enthält, die an das Subprogramm übergeben werden sollen.
- Der vierte Parameter enthält die Adresse der Tabelle, die die Format-/Längenangaben der Parameter enthält, die an das Subprogramm übegeben werden sollen.

Reihenfolge, Format und Länge der Parameter im aufrufenden Programm müssen exakt übereinstimmen mit Reihenfolge, Format und Länge der Felder im `DEFINE DATA PARAMETER`-Statement des Subprogramms. Die Namen der Felder können im aufrufenden Programm und im aufgerufenen Subprogramm unterschiedlich sein.

Beispiel: Aufruf eines Natural-Subprogramms von einem 3GL-Programm

Die folgenden Beispiele im *samples\sysexuex*-Unterzeichnis des Natural-Verzeichnisses zeigen, wie Sie ein Natural-Subprogramm von einem 3GL-Programm aufrufen können:

- *MY3GL.NSP* (für das Hauptprogramm),
- *MY3GLSUB.NSN* (für das Subprogramm),
- *MYC3GL.C* (für die "C"-Function).

58 Betriebssystemkommandos aus einem Natural-Programm absetzen

- Syntax 540
- Parameter 540
- Parameter-Optionen 540
- Return-Codes 541
- Beispiele 541

Mit dem Natural-User-Exit SHCMD können Sie direkt aus einem Natural-Programm heraus ein Betriebssystemkommando absetzen.

Syntax

```
CALL 'SHCMD' 'command' ['option']
```

Parameter

command	*command* wird vom Betriebssystem ausgeführt. Um Kommandos auszuführen (zum Beispiel DIR oder DEL) müssen Sie auch den Systemkommandointerpreter angeben. Weitere Informationen siehe ***Beispiele*** unten.
option	*option* beschreibt wie das Kommando ausgeführt werden soll. Dieser Parameter ist optional. Die folgenden Optionen stehen zur Verfügung: ■ ASYNCH ■ NOSCREENIO ■ SYNCH (Standard) Siehe *Parameter-Optionen* unten.

Parameter-Optionen

Die folgenden Optionen stehen zur Verfügung:

Option	Beschreibung
ASYNCH	Natural wartet nicht bis das Kommando vollständig ausgeführt wurde. Diese Art der Verarbeitung wird asynchrone Verarbeitung genannt.
NOSCREENIO	Diese Option wird benutzt, um die von dem Kommando generierte Ausgabe zu verstecken. Die versteckte Ausgabe wird an das Null-Device umgeleitet.
SYNCH	Natural wartet bis das Kommando vollständig ausgeführt wurde. Diese Art der Verarbeitung wird synchrone Verarbeitung genannt. Dies ist der Standard.

Anmerkung: Die Optionen ASYNCH und SYNCH dürfen nicht gleichzeitig gesetzt werden.

Return-Codes

Die folgenden Return-Codes stehen zur Verfügung:

Return-Code	Beschreibung
0	Kommando wurde erfolgreich ausgeführt.
4	Unerlaubter SHCMD-Parameter wurde angegeben.
Alle anderen	Betriebssystemabhängiger Fehlercode.

Beispiele

Das Betriebssystemkommando `DIR` ausführen, um den Inhalt eines Verzeichnisses anzuzeigen:

```
CALL 'SHCMD''CMD.EXE /C DIR'
```

Den Return-Code mit der Natural-Funktion `RET` abfragen:

```
RESET rc (I4)
CALL 'SHCMD''CMD.EXE /C DIR'
rc = RET( 'SHCMD' )                    /* retrieve return code
IF rc <> 0 THEN                        /* in case of an error
DISPLAY "Error occurred during SHCMD" /* display an error message
```

Ein Kommando, das Leerzeichen enthält, wird in Anführungszeichen angegeben. Im folgenden Beispiel wird Microsoft Excel aufgerufen:

```
RESET #cmd (A253)
MOVE '"C:\Program Files\Microsoft Office\Office\EXCEL.EXE"' to #cmd
CALL "SHCMD" #cmd "ASYNCH"
```

In diesem Fall muss der Parameter `TQ` (Anführungszeichen übersetzen) auf `OFF` gesetzt werden, damit die Anführungszeichen nicht entfernt werden.

Um vom Parameter `TQ` unabhängig zu sein, können Sie den hexadezimalen ASCII-Code für Anführungszeichen (H'22') am Anfang und Ende des Kommandos einfügen:

```
RESET #cmd (A253)
MOVE H'22' - "C:\Program Files\Microsoft Office\Office\EXCEL.EXE" - H'22' to #cmd
CALL "SHCMD" #cmd "ASYNCH"
```

59 Statements für den Internet- und XML-Zugriff

- Verfügbare Statements 544
- Weitere Informationsquellen 545

Dieses Kapitel gibt einen Überblick über die Natural-Statements für den Zugriff auf Internet und XML und enthält ein Literaturverzeichnis.

Um diese Statements vollständig nutzen zu können, ist eine gründliche Kenntnis der zugrundeliegenden Kommunikationsstandards erforderlich.

Verfügbare Statements

Die folgenden Natural-Statements stehen zum Zugriff auf das Internet und XML zur Verfügung:

- REQUEST DOCUMENT
- PARSE XML

REQUEST DOCUMENT

Dieses Statement ermöglicht es Ihnen, das HTTP-Protokoll zu verwenden.

Das folgende Beispiel zeigt, wie dieses Statement benutzt werden kann, um auf ein extern vorliegendes Dokument zuzugreifen:

```
REQUEST DOCUMENT FROM
"http://bolsap1:5555/invoke/sap.demo/handle_RFC_XML_POST"
WITH
USER #User PASSWORD #Password
DATA
NAME 'XMLData' VALUE #Queryxml
NAME 'repServerName' VALUE 'NT2'
RETURN
PAGE #Resultxml
RESPONSE #rc
```

Weitere Informationen siehe `REQUEST DOCUMENT` in der *Statements*-Dokumentation.

PARSE XML

Das `PARSE XML`-Statement ermöglicht es Ihnen, XML-Dokumente von einem Natural- Programm zu parsen.

Weitere Informationen siehe `PARSE XML` in der *Statements*-Dokumentation.

Weitere Informationsquellen

Die folgende Aufstellung enthält Hinweise auf weitere nützliche Informationsquellen:

- Schulungskurse
- Nützliche Links

Schulungskurse

Die Schulungsabteilung der Software AG bietet spezielle Schulungskurse zu diesem Thema an. Ausführlichere Informationen siehe ServLine24 unter ***http://servline24.eur.ad.sag/public/***.

Oder wenden Sie sich an Ihre örtliche Software AG-Vertretung wegen spezieller Schulungen vor Ort.

Nützliche Links

Allgemeine Informationen finden Sie auf den folgenden Web-Seiten:

- World Wide Web Consortium (W3C): ***http://www.w3.org/***
- Extensible Markup Language (XML): ***http://www.w3.org/XML/***
- HyperText Markup Language (HTML) Home Page: ***http://www.w3.org/MarkUp/***
- W3 Schools: ***http://www.w3schools.com/***

60 Portierbare generierte Natural-Programme

- Kompatibilität ... 548
- Hinweise zum Endian-Mode ... 548
- ENDIAN-Parameter ... 549
- Natural-GPs übertragen ... 549
- Portierbare FILEDIR.SAG- und Fehlermeldungsdateien ... 551

Ab Natural Version 5 sind generierte Natural-Programme (GPs) auf alle UNIX-, OpenVMS- und Windows-Plattformen portierbar.

Kompatibilität

Ab Natural Version 5 ist eine Source, die auf einer von Natural unterstützten UNIX-, OpenVMS- oder Windows-Plattform katalogisiert wurde, auf allen von Natural unterstützten Open-Systems-Plattformen ohne Neu-Kompilierung ausführbar.

Natural-Anwendungen, die mit Natural Version 4 or Natural Version 3 erstellt wurden, können mit Natural Version 5 oder höher ausgeführt werden, ohne dass diese Anwendungen erneut katalogisiert werden müssen (Aufwärtskompatibilität). In diesem Fall ist die portierbare GP-Funktionalität nicht verfügbar. Um diese Funktionalität und andere Verbesserungen nutzen zu können, ist das Katalogisieren mit Natural Version 5 oder höher erforderlich.

Kommandoprozessor-GPs sind nicht portierbar. Das GP-Portierbarkeitsmerkmal steht nicht für Großrechnerplattformen zur Verfügung. Das bedeutet, dass die auf Großrechnern erstellten GPs nicht ohne Neu-Kompilierung auf UNIX-, OpenVMS- und Windows-Plattformen und umgekehrt ausgeführt werden können.

Hinweise zum Endian-Mode

Ab Natural Version 5 verhält sich Natural folgendermaßen: Je nachdem, auf welcher UNIX-, OpenVMS- oder Windows-Plattform Natural läuft, berücksichtigt es die Byte-Reihenfolge, in der mehrere Bytes umfassende Zahlen im GP gespeichert werden. Die Formate mit Zwei-Byte-Reihenfolge werden als „Little Endian“ und „Big Endian“ bezeichnet.

- „Little Endian“ bedeutet, dass das niederwertige Byte der Zahl unter der niedrigsten Adresse und das höherwertige Byte unter der höchsten Adresse im Speicher abgelegt werden (zuerst kommt der Little Endian).
- „Big Endian“ bedeutet, dass das höherwertige Byte der Zahl unter der niedrigsten Adresse im Speicher abgelegt wird (zuerst kommt der Big Endian).

Die UNIX-, OpenVMS- und Windows-Plattformen verwenden beide Endian-Formate: Intel-Prozessoren und AXP-Computer verwenden die „Little-Endian“-Byte-Reihenfolge, andere Prozessoren wie HP-UX, Sun Solaris oder RS6000 verwenden das "Big-Endian"-Format.

Falls erforderlich, wandelt Natural ein portierbares GP automatisch in das Endian-Format der Ausführungsplattform um. Diese Endian-Umwandlung erfolgt nicht, wenn das GP schon im Endian-Format der Plattform erzeugt worden ist.

ENDIAN-Parameter

Um die Performance bei der Ausführung von portablen GPs zu erhöhen, wurde der Natural-Profilparameter ENDIAN eingeführt. ENDIAN bestimmt das Endian-Format, in dem ein GP bei der Kompilierung generiert wird.

DEFAULT	Das Endian-Format der Maschine, auf der das GP generiert wird.
BIG	Big-Endian-Format (höherwertiges Byte zuerst).
LITTLE	Little-Endian-Format (niederwertiges Byte zuerst).

Die Werte DEFAULT, BIG und LITTLE können alternativ angegeben werden, wobei DEFAULT der Standardwert ist.

Der ENDIAN-Parameter kann folgendermaßen gesetzt werden:

- als Profilparameter in der Natural Configuration Utility,
- als Startup-Parameter,
- als Session-Parameter oder mit dem Systemkommando GLOBALS.

Natural-GPs übertragen

Um portierbare GPs auf verschiedenen Plattformen (UNIX, OpenVMS- und Windows) verwenden zu können, müssen die generierten Natural-Objekte auf die Zielplattform übertragen werden oder sie müssen von der Zielplattform aus erreichbar sein, z.B. über NFS.

Für die Verteilung von Natural-GPs oder von ganzen Natural-Anwendungen wird der Natural Object Handler empfohlen. Dabei werden die Objekte in der Source-Umgebung in ein Work File geladen, danach das Work File in die Zielumgebung übertragen und dann die Objekte aus dem Work File entladen.

Um Ihre generierten Natural-Objekte über Open-Systems-Plattformen zu verteilen

1 Starten Sie den Natural Object Handler. Laden Sie alle gewünschten katalogisierten Objekte in ein Work File des Typs PORTABLE.

Falls Fehlermeldungen benötigt werden, können diese ebenfalls in das Work File geladen werden.

Wichtig: Der angegebene Work-File-Typ muss PORTABLE sein, weil Natural bei diesem Typ eine automatische ENDIAN-Formatumwandlung eines Work Files vornimmt,

wenn dieses auf eine andere Maschine übertragen wird. Siehe auch Informationen zum Work-File-Typ in der Beschreibung des `DEFINE WORK FILE`-Statements in der *Statements*-Dokumentation.

2 Übertragen Sie das Work File in die Zielumgebung. Je nach Übertragungsmedium (Netzwerk, CD, Diskette, Band, Email, Download usw.) kann es sinnvoll sein, ein komprimiertes Archiv (z.B. ZIP-Datei) oder eine Codierung/Decodierung (z.B. mit UUENCODE/UUDECODE) vorzunehmen. Beim Kopieren per FTP muss die Übertragungsart binär sein.

Anmerkung: Je nach verwendetem Übertragungsverfahren kann es nötig, sein, das Record-Format und die Attribute oder die Blockgröße des zu übertragenen Work Files an die Zielumgebungsplattform anzupassen, bevor die Ladefunktion benutzt wird. Das Work File sollte auf der Zielplattform dasselbe Format und dieselben Attribute haben wie ein Work File desselben Typs, das auf der Zielplattform selbst erstellt wurde. Falls eine Anpassung nötig sein sollte, können Sie dazu Betriebssystem-Tools benutzen.

3 Starten Sie den Natural Object Handler in der Zielumgebung. Als Work-File-Typ wählen Sie PORTABLE. Laden Sie die Natural-Objekte und -Fehlermeldungen aus dem Work File.

Weitere Informationen zur Benutzung des Natural Object Handler finden Sie im Teil *Object Handler* in der *Utilities*-Dokumentation.

Weitere Informationen, wie Sie eine Anwendung von einer Natural-Entwicklungs-Workstation auf eine Natural-Laufzeit-Workstation portieren können, finden Sie im Abschnitt *Porting Procedure Overview* in der *Operations*-Dokumentation.

Außer dem oben beschriebenen bevorzugten Verfahren gibt es noch verschiedene andere Möglichkeiten, um Natural-GPs oder sogar ganze Libraries oder Teile davon mittels Betriebssystem-Tools und verschiedenen Übertragungsverfahren zu verschieben oder zu kopieren. Damit die Objekte unter Natural ausführbar sind, müssen sie in all diesen Fällen in die Natural-Systemdatei FUSER importiert werden, damit die *FILEDIR.SAG*-Struktur angepasst wird. Informationen zu Verzeichnis FNAT bzw. FUSER finden Sie im Abschnitt *System Files FNAT and FUSER* in der *Operations*-Dokumentation.

Sie können eines der folgenden Verfahren anwenden:

- Sie können die Import-Funktion der Natural-SYSMAIN-Utility benutzen.
- Sie können die FTOUCH-Utility benutzen. Diese Utility können Sie außerhalb von Natural aufrufen und benutzen.
- Darüber hinaus können Sie auch Dateien vom Windows Explorer in die Natural-Umgebung importieren, indem Sie Drag-and-Drop oder die Menübefehle **Cut**, **Copy** und **Paste** benutzen. Das heißt, wenn Sie Zugriff auf die Natural-Objekte haben, die Sie über den Windows Explorer importieren wollen, können Sie Drag-and-Drop oder **Cut**, **Copy** und **Paste** benutzen, und die *FILEDIR.SAG*-Datei wird dabei automatisch aktualisiert. Weitere Informationen zum Kopieren,

Verschieben und Importieren von Objekten finden Sie im Abschnitt *Natural-Objekte verwalten* in der Dokumentation *Natural Studio benutzen*.

Dasselbe gilt, wenn der direkte Zugriff von einer Zielplattform auf generierte Objekte in einer solchen Umgebung möglich ist, z.B. über NSF, Netwerk-File-Server usw. In diesem Fall müssen die Objekte ebenfalls importiert werden.

Portierbare FILEDIR.SAG- und Fehlermeldungsdateien

Ab Natural Version 6.2 sind die Datei *FILEDIR.SAG* und die Fehlermeldungsdateien plattformunabhängig. Somit ist es möglich, FUSER-Systemdateien gemeinsam von unterschiedlichen Open-Systems-Plattformen aus zu nutzen. Zum Beispiel ist es möglich, unter Verwendung von Kopierfunktionen des Betriebssystems mehrere Natural-Libraries von einer Open-Systems-Plattform zu einer anderen zu kopieren. Die gemeinsame Nutzung von FNAT-Systemdateien wird jedoch nicht empfohlen. Weitere Informationen zur portierbaren *FILEDIR.SAG*-Datei finden Sie im Abschnitt *Portable Natural System Files* in der *Operations*-Dokumentation.

61 Introduction to Event-Driven Programming

- What is an Event-Driven Application?
- GUI Development Environments
- GUI Design Tips
- Tasks Involved in Creating an Application
- Tutorial
- Basic Terminology

For detailed information on event-driven programming, see *Event-Driven Programming Techniques*.

62 What is an Event-Driven Application?

- Introduction 556
- Program-Driven Applications 557
- Event Driven Applications 558
- What is Happening Here? 559
- Writing Event-Driven Code 559
- Components of an Event Driven Application 560

Introduction

Event-driven applications represent a new approach to development in addition to the program-driven approach. Natural offers you both. Event-driven programming allows the application to be driven by input received through the graphical user interface.

In program-driven applications, the application controls the portions of code that execute - not an event. Execution starts with the first line of executable code and follows a defined pathway through the application, calling additional programs as instructed in the predetermined sequence.

In event-driven programming, the user's action or a system event triggers the code attached to that event. Thus, the order in which your code executes depends on which events occur, which in turn depends on what the user does. This is the essence of graphical user interfaces and event-driven programming: The user is in charge, and the code responds. Even though event-driven programming is possible in character-oriented interfaces, it is more common in graphical user interfaces.

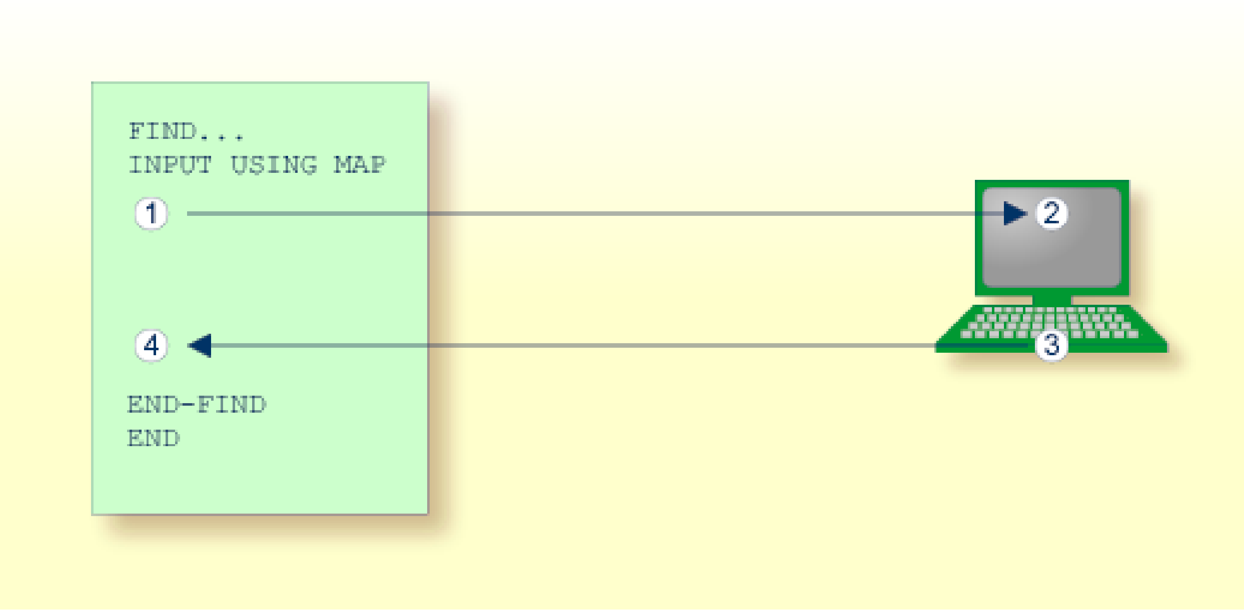
Because you cannot predict what the user will do, your code must make a few assumptions when it executes. For example, the application might assume that the user added text to an edit-area control before pressing the **OK** button.

When you must make assumptions, you should try to structure your application so that these assumptions are always valid. For example, to ensure the user added text, you can disable the button and enable it only when the change event occurs for the edit area control.

Your code can trigger additional events as it performs certain operations. For example, moving the slider in a scroll bar control triggers the change event.

The following diagrams illustrate the difference between program-driven and event-driven applications.

Program-Driven Applications

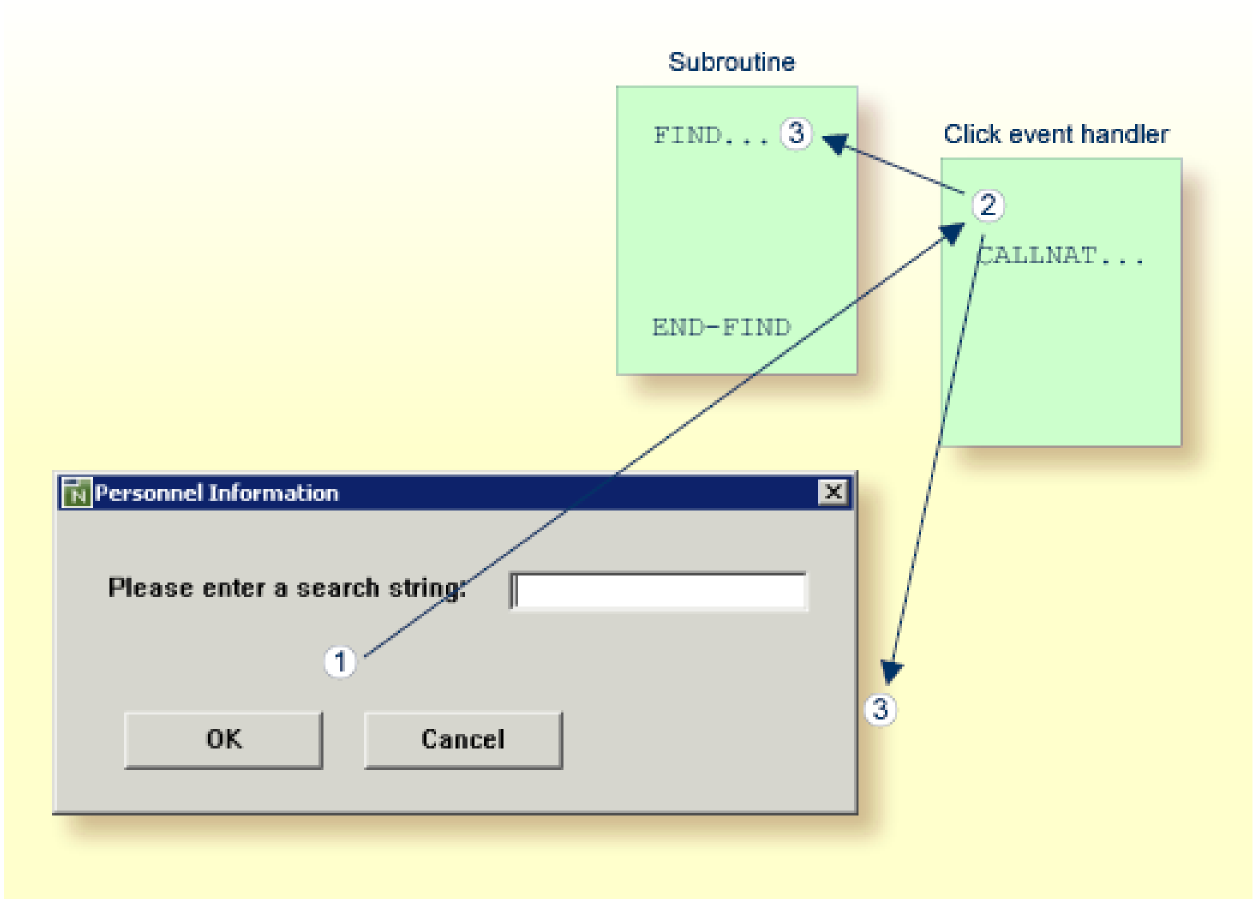


In typical program-driven applications, the following sequence of steps applies:

1. The program sends a screen to the terminal.
2. The user reacts by filling in the data fields.
3. The user then presses ENTER or a function key.
4. The program then decides whether or not the user's entries are valid.

If the data are valid, it processes the results until it reaches an `END` statement.

Event Driven Applications



In typical event-driven applications, the following sequence of steps applies:

1. The user requests an action on the screen.
2. The event handler code reacts in the background according to the context.
3. If certain conditions are fulfilled, the executed event handler code triggers other Natural code (here: a subroutine) or returns control to the screen.

In the program-driven approach, the user interacts with the code through the ENTER and function keys, the user of an event-driven application triggers specific pieces of code (event handlers). Typically, an event-driven application is not executing any code when waiting for user input; in the same situation, the program-driven application might be processing an INPUT statement.

What is Happening Here?

Graphical user interface programs require you to write programs that react to isolated events initiated by the user.

An event is an action recognized by a dialog or a dialog element. Event-driven applications execute code in response to an event. Each dialog or dialog element has a predefined set of events. If one of these events occurs, Natural invokes the code in the associated event handler.

You decide if and how the dialogs and dialog elements in your application respond to a particular event. When you want a program to respond to an event, you write event code for that event.

Writing Event-Driven Code

For each dialog or dialog element you create, Natural predefines a set of events to which your program (event handler) can respond. It is easy to respond to events: dialogs and dialog elements have the built-in ability to recognize user actions and execute the code associated with them.

You do not have to write code for all events. When you do want a dialog object to respond to an event, you write event code that Natural executes in response to that event.

In a typical event-driven application, the following series of actions takes place:

- A dialog or dialog element recognizes an action as an event. The action can be caused by the user (such as a click or keystroke).
- If there is event code corresponding to the event, it is executed.
- The application waits for the next event.

The event code you write to respond to events can perform calculations, get input, and manipulate parts of the interface. Using Natural, you manipulate dialogs or dialog elements by changing the values of their attribute settings.

Vorsicht: Avoid creating cascading events in your code caused by events occurring repeatedly. For example, when the user drags the slider in the scroll-bar control, the current `SLIDER` attribute setting is automatically changed and the change event is triggered. If the code attached to the change event also changes the current `SLIDER` attribute setting, then the change event is triggered again, the current `SLIDER` attribute setting is again adjusted, the change event is once again triggered, and so on. At this rate, you quickly run out of memory.

Components of an Event Driven Application

The following topics are covered below:

- Dialogs
- Dialog Elements
- Attributes
- Event Handlers
- Data Areas - Global, Local, Parameter
- Inline Subroutines

Dialogs

The dialog is the central Natural object in an event-driven application. An event-driven application is started by running or executing the base dialog. This may open other dependent dialogs when the OPEN DIALOG statement is specified. As opposed to program-driven applications, these dialogs are usually modeless, that is, all open dialogs can be processed concurrently by the end user. The application terminates when the base dialog is closed.

You create a dialog with the dialog editor. Just like the map editor, the dialog editor assembles a Natural object from the specification of the dialog window and its dialog elements, the global data area (GDA), the local data areas (LDAs), the parameter data areas (PDAs), the subroutines and the specified event handler sections.

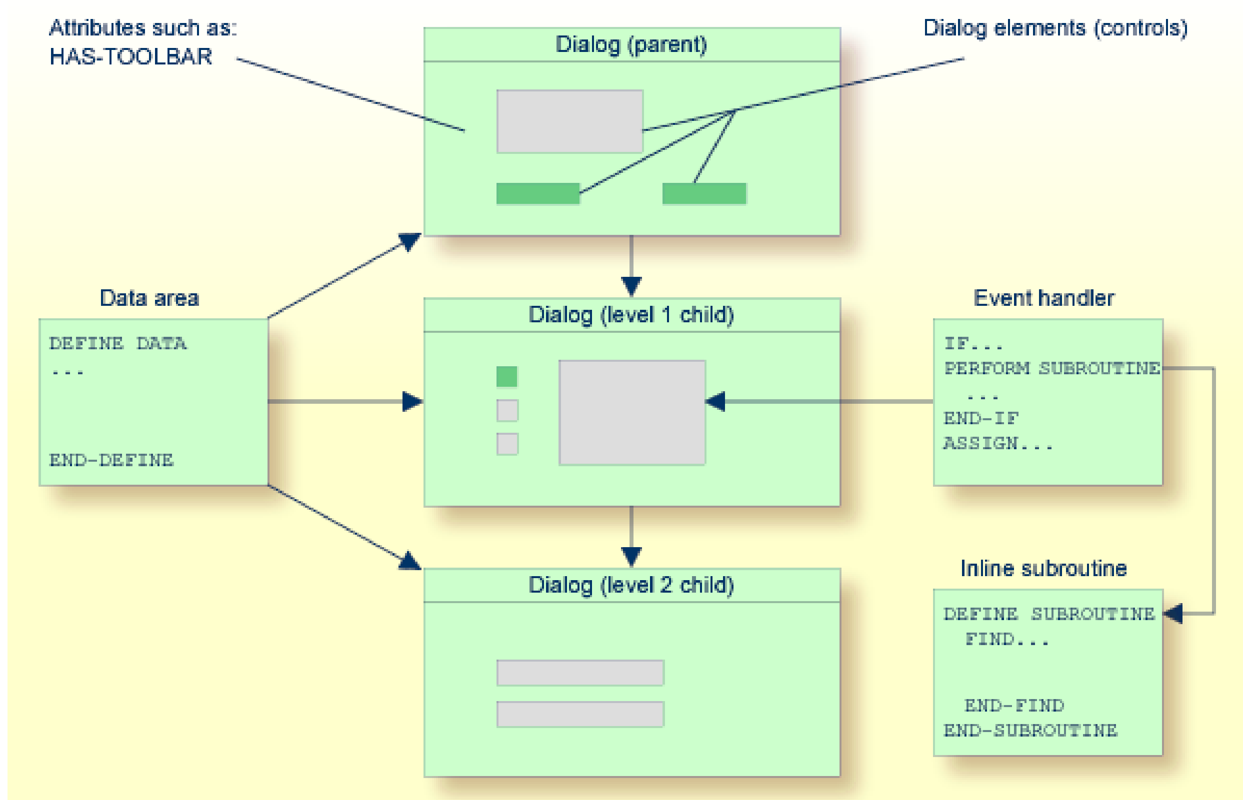
At runtime of the dialog, there is a difference between the runtime instance identified by the system variable *DIALOG-ID and the GUI instance (handle) of the dialog window (the default handle name is #DLG$WINDOW).

Whenever you want to work with more than one dialog in your application, you must decide how the base dialog window relates to the other dialogs. First you have to decide whether the application should be MDI (Multiple Document Interface) or not.

If you have opted for an MDI application, the base dialog must be of the type „MDI frame window" and the dependent dialogs must be of the type „MDI child window" and „Standard window".

If you have opted for non-MDI, the application may contain only dialogs of the type „Standard window".

Dialogs of type „Standard window“ can have the styles "Popup", "Modal" or "Dialog Box".



Dialog Elements

Almost all dialog elements are graphical elements inside a dialog that allow the end user to interact with the event-driven application. After a dialog has been opened with the dialog editor and its attributes have been set (see below), the programmer will go on to „draw“ the dialog elements inside the window; usually, this comprises a menu control, possibly a toolbar, and other elements, such as push-button controls, input-field controls.

„Drawing“ a dialog element means that you select the type of dialog element from the dialog editor's menu or toolbar, and use the mouse to place it at the desired location. It is also possible to define a grid where the dialog elements can be placed more conveniently by aligning them to the grid.

Attributes

Attributes are the properties of dialogs and dialog elements. After creating a dialog or dialog element, you double-click with the mouse on it and the window with the corresponding attributes appears. You can then set the attributes to a value; if not, they remain at the system default value. The attributes window also contains a push-button control that opens up the event handler window.

Event Handlers

The event handlers represent the Natural code that is triggered when an event occurs. A click event occurs, for example, when the end user clicks on a push-button control. Inside the event handler window, you must first select the type of event from the list of events available for the dialog or dialog element (the one whose attributes have just been set). Then, the code window is enabled and Natural code can be entered.

Data Areas - Global, Local, Parameter

- A global data area (GDA) is used to share data fields between Natural objects within the application. One GDA per application may be specified.
- A local data area (LDA) contains the data fields private to the dialog.
- A parameter data area (PDA) is always present in dialogs. It is used to pass parameters to a dialog in the `OPEN DIALOG` or `SEND EVENT` statements. In these statements, parameters are passed either by specifying their name (`WITH` clause), or by listing parameters one after the other. You can use the dialog editor PDA window to type in your PDA in free-form style or to include PDAs defined externally.

Inline Subroutines

An inline subroutine defines standard code to be used for a frequently needed task called by a number of event handlers. You access an inline subroutine window via the "Inline Subroutines" push button control in the dialog window.

63 GUI Development Environments

To understand the functions of Natural, you must first understand the environment in which it runs.

A graphical user interface (GUI) environment differs from a traditional mainframe environment in at least two important ways:

- Applications share screen space. A Natural application runs in a group of one or more windows and rarely occupies the full screen.
- Applications share computing time. An application cannot run continually, or if it does, it must run in the background.

Using Natural, your applications share computing time and other resources (such as the clipboard). An event-driven application consists of dialogs and dialog elements that wait for a particular event to happen.

While your application is waiting to execute an event, it remains on the desktop (unless the user closes the application). In the meantime, the user can run other applications, resize windows, or customize system settings (such as color). However, your code is always present, ready to be activated when the user returns to your application.

64 GUI Design Tips

- Introduction 566
- Do Your Research 566
- Screen Design 567
- Menu Design 568
- Color Usage 569
- Consistency Check 569

Introduction

Designing the screens for a GUI application requires different knowledge than designing the 3270 screens for a mainframe. Why is it different?

It is different, because GUI applications put the users in control; these applications are non-modal and unstructured. The users choose the order in which they access windows, and fields within the windows. Traditional database applications often require the users to perform operations in a specific order; these applications are form-oriented and structured.

Designing a GUI screen is also different, because the GUI interface has different capabilities than a traditional mainframe interface. You can design windows that incorporate dialog elements, such as push button controls and list box controls. As you design your GUI windows, which are called dialogs in event-driven Natural, you define the font type and size of the text, the background and foreground colors, and the size of each window.

The following sections provide some tips for effective GUI design.

Do Your Research

- Spend a few hours with your users before prototyping.

  A couple of sessions with your users to iron out their needs, likes, and dislikes is enough to give you to a good basis for beginning your design.

- Take some ideas from existing GUI designs.

  Save time by not re-inventing the GUI. Try out other GUIs with an eye for what works and what does not. Consistency within GUIs helps users learn to use new applications, improves efficiency, and reduces training costs. Get user feedback on existing GUI applications - listen to their likes and dislikes rather than develop a prototype that replicates the weaknesses of poor GUI design.

- Develop your ideas on paper before spending time developing the application online.

  It is faster for you to run through a number of screen design options for your main windows on paper before spending time to create multiple prototypes online. It is quicker than coding and you do not become attached to poor designs.

  If you include your users in the development process, they can quickly comment about their needs and likes before the application is installed in the system. Try to use a paper prototype before reaching for the online development tool.

Screen Design

- Design multiple windows for related subject matter.

  Unlike designing for 3270 monitors, where you try to maximize the number of fields per screen, GUI screens are better designed using subwindows. You can, for example, have the essential fields in the main window, and all optional or supplemental information stored in one or more subwindows. Subwindows can include choices, such as drop-down lists, for the user to browse through if they do not know the information to input into the main window. Messages and field-dependent information are more effectively presented in supplemental windows than in the main window.

- Design clear, uncluttered windows.

  Avoid cluttering your windows with more than three colors, multiple graphics, and a variety of shapes. Balance your objects on the screen with lots of white space so users are not overwhelmed by variety and distracted by the presentation. Try to keep shapes and objects to a minimum and the number of colors low.

- Design accessible, not overwhelming, windows.

  Multiple fonts, font sizes, font types or families, and color schemes can overwhelm your users, making your application seem inaccessible to them. Use a maximum of three fonts, font sizes, and font types per window. Avoid using italics and serif fonts because they often break up on the screen. Use color sparingly. Neutral colors are kindest to your users eyes. Though vibrant reds and greens are very eye-catching, remember that your users spend a lot of their day working in the windows you design.

- Design for both keyboard and mouse use.

  Some users prefer using the keyboard and memorize the short cut commands, while other users are more comfortable using the mouse. Each action should be accessible by both the mouse and the keyboard.

- Design the windows according to your users' needs.

  Though it is tempting to create fabulous-looking screens with lots of functionality, if your users do not use it, it is of no value. Remember that you are designing the application for your users to get a job done, not for you to experiment with all the functionality you have available. First find out what your users need, then tailor your design to meet their needs. You design screens with different purposes in different ways. If you want to prompt the user, you use a conversational style; if you want the user to enter values from a form, you use a data-entry style.

Conversational Screens

- Design conversational screens with field prompts.

  In a conversational-based style, users enter data from a conversation (travel reservations, for example). Conversational-based styles, in which the user relies on the screen for prompting, can be rich with labels, hints, instructions, and even questions for the users to ask their clients.

Data-Entry Screens

- Design data-entry screens with terse labels.

  In a form-based style, users enter data from a form. Each line on the input screen must match a line on the form - and the lines must be in the same order. To maintain a line-for-line correspondence, you can abbreviate labels. Headings and instructions are kept to a minimum. The only purpose of labels is to help users find their places again after interruptions.

Menu Design

The following three criteria are recommended for designing menus.

- Organize menus using the conventions defined by the operating system on which your users run the application. Microsoft Windows, for example, recommends certain menus (**File**, **Edit**, and **View**, for example), options on menus (**Cut**, **Copy**, and **Paste** on the **Edit** menu, for example), and a particular order of the menus on the menu bar (**Help** always appears at the right margin, for example).
- Arrange menus by frequency of use and decide this information through observation or usability testing.

  Anticipate whether usage changes as users become more expert. Watch that this does not violate conventions established for the operating system.
- List menu items alphabetically. Remember to follow the operating-system conventions and user recommendations for frequency of using menu items.

Color Usage

- Be as conservative as possible with color.

  Humans can remember the meaning of no more than five colors at a time, plus or minus two.
- Use color as an additional signal, not as the primary signal.

  Using bright red text to warn a user is not enough; add a warning tone. Eight percent of all males are red-green color-blind and may not notice the red text.
- On charts, do not use colors without adding a secondary key (for example, a broken or solid underline).

  Users with black and white monitors must be able to understand the key without the benefit of color. Also, most users do not have color printers.

Consistency Check

- Be consistent throughout the application.

  Do not change fonts, colors, or shapes for related subjects. For example, design all the **OK** buttons in an application with the same shape, size, color, and font. If related objects are presented in different ways, users cannot use the visual clues, taking them longer to become comfortable with the application. Present similar actions in a similar way, using the same font, color, and size for related buttons.
- Adopt a naming convention (and stick with it throughout the application).

  While traditional programs tend to have one large program you modify for a name change, object-oriented programs have numerous pieces of event code that you must edit individually every time you make a name change. When you design GUI applications, you must be much more rigorous about sticking to naming conventions. This avoids a lot of cleaning up time later.

65 Tasks Involved in Creating an Application

There are a number of main tasks you perform to create an application in event-driven Natural. The order in which they are explained in this section is the typical order in which you perform them. However, this sequence is not inflexible. For example, you may very well test a dialog several times in the process of designing it, and you will no doubt save your work more often during the development process.

- Decide whether your application is „Multiple Document Interface“ or „Single Document Interface“.
- Create one or more dialogs.
- Set the attributes of the dialog(s).
- Create and place dialog elements in the dialog(s).
- Set the attributes of the dialog elements.
- Define the tab order in each of the dialogs (from the **Dialog** menu, choose **Control Sequence**).
- Save the dialog(s) to a name.
- Define the global data area.
- Define the local data area(s).
- Write event handler code for the dialog(s).
- Write inline subroutines for the dialog(s).
- Write event handler code for the dialog elements.
- Stow the dialog(s).
- Test (check and run) the dialog(s).
- Execute the application.

The **tutorial** in the next section introduces you to the most frequently performed tasks.

66 Tutorial

- Creating a Dialog 574
- Assigning Attributes to the Dialog 575
- Creating Dialog Elements Inside the Dialog 577
- Assigning Attributes to the Dialog Elements 579
- Creating the Application's Local Data Area 580
- Attaching Event Handler Code to the Dialog Element 581
- Checking, Stowing and Running the Application 582

This document is a simple tutorial that demonstrates how to add the components of an event-driven application one after the other. The tutorial describes how to develop a small sample application consisting of one dialog. The application you will create is a degressive depreciation calculator.

You can use this calculator, for example, to find out the value of your car by entering how much the car was worth when you bought it, how many years you have owned it, and the percentage by which the value of the car decreases each year.

You can save your application at any stage, allowing you to interrupt the tutorial and continue at a later time where you left.

To develop the sample application

1. Create a new dialog (represented by a window).
2. Assign the attributes to your dialog (decide the window's settings).
3. Create the dialog elements in the dialog (decide how the user can interact).
4. Assign the attributes to your dialog elements (decide attribute settings).
5. Create the application's local data area (define the variables that allow the event handler to use the end user's numeric input).
6. Attach event handler code to the dialog element (decide what happens at runtime when the user interacts).
7. Check, stow and run the application.

Apart from creating the local data area, this is the minimal number of steps required to create any event-driven application.

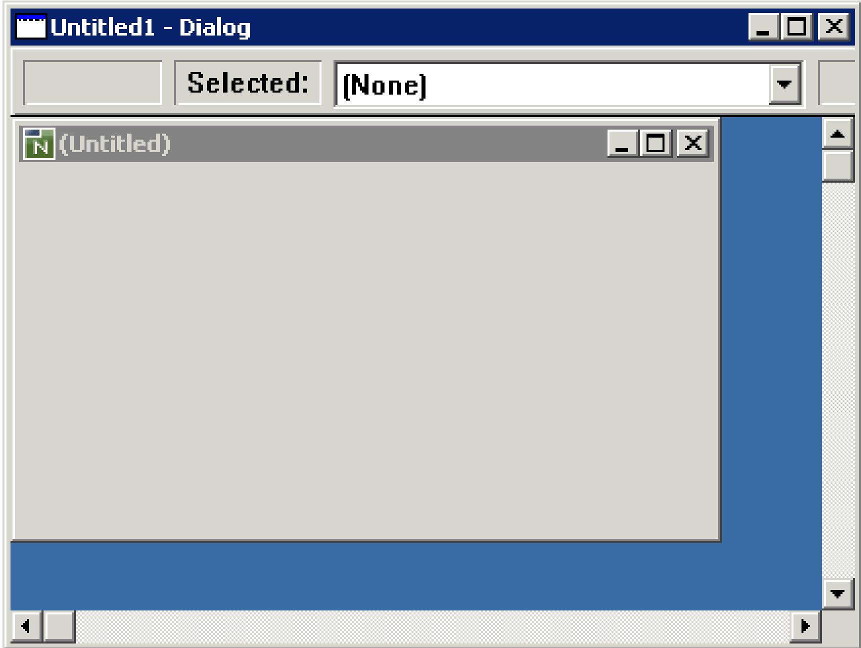
The above steps are described in detail in the following topics:

Creating a Dialog

To create a new dialog

1. Invoke Natural.
2. From the **Object** menu, choose **New > Dialog**.

The Natural window displays the menus and the toolbar for the dialog editor. It displays an editing window called "Untitled1 - Dialog". You can resize this editing window.



The editing window contains the new dialog window, titled "(Untitled)". You can also resize this new dialog window, or use the editing window's scroll bars.

Assigning Attributes to the Dialog

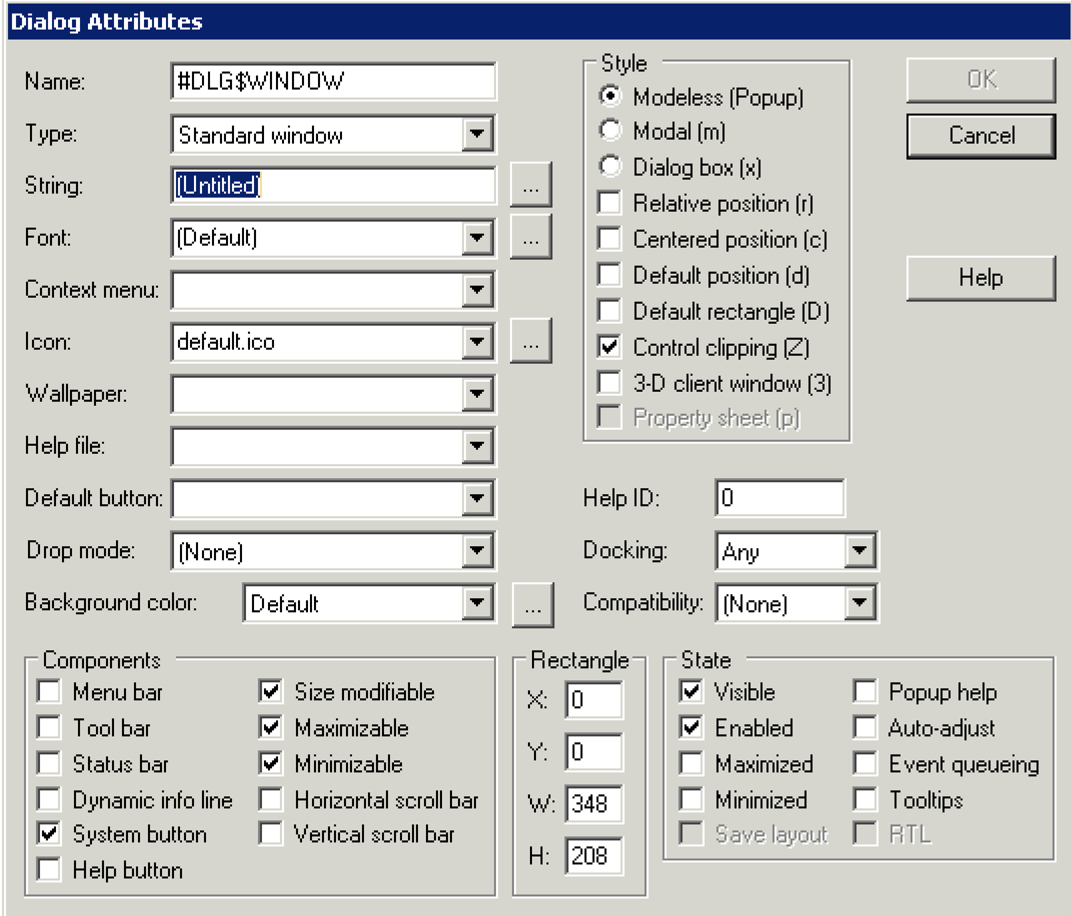
To assign attributes to the dialog

1 From the **Dialog** menu, choose **Attributes**.

Oder:

Double-click inside the dialog window.

The **Dialog Attributes** dialog box appears.



2 With the cursor in the **String** text box, type in the new dialog window's title: "Degressive Depreciation".

3 From the **Background color** drop-down list box, select the desired color, for example **Gray**.

4 Choose the **OK** button.

The **Dialog Attributes** dialog box closes.

You have set the attribute `STRING` to the value "Degressive Depreciation" and the attribute `BACKGROUND-COLOUR-NAME` to the value of your desired color, for example `GRAY`.

Creating Dialog Elements Inside the Dialog

To create the dialog elements inside the dialog

1. From the **Tools** menu, choose **Options**.

   The **Options** dialog box appears.

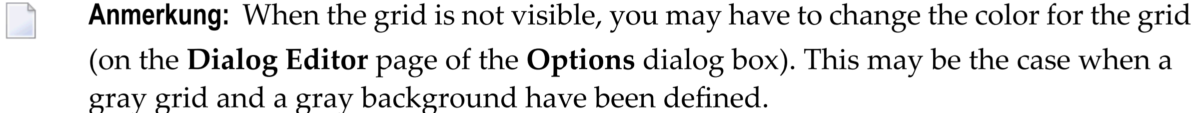
2. Select the **Dialog Editor** page.

3. Make sure that the **Display grid** check box is selected and select the **Lines** option button.

   This decides the way your grid will be displayed.

4. Choose the **OK** button to confirm the change.

   The grid now helps you position and align the dialog elements.



Anmerkung: When the grid is not visible, you may have to change the color for the grid (on the **Dialog Editor** page of the **Options** dialog box). This may be the case when a gray grid and a gray background have been defined.

5. From the **Insert** menu, choose **Text Constant**.

   Oder:

   Choose the toolbar button representing a text constant control.

6. Move the cursor to the upper left corner of the dialog window.

   Ensure that the editor window's status bar displays an x and a y value of less than 50. Note that at this time, the text constant control's width and height has an undefined value.

7. Click to fix the text constant control's position.

   A grey rectangle representing the dialog element appears, surrounded by small black squares. At the same time, the status bar indicates that `#TC-1` is selected.

8. Point to one of the small black squares.

   The cursor shape now indicates the direction in which you can resize the text constant control.

9. Resize `#TC-1` to a width of about 200.

10. Make sure that the text constant control is selected.

11. From the **Edit** menu, choose **Copy**.

12. From the **Edit** menu, choose **Paste**.

    A new text constant control `#TC-2` is created on top of `#TC-1`.

13 Move the new text constant control to a position below the first one by clicking and dragging via the mouse, or via the keyboard arrow keys with the SHIFT key held down.

14 Create another text constant control below the previous text control (in the same way).

15 From the **Insert** menu, choose **Input Field**.

Oder:

Choose the toolbar button representing an input field control.

16 Position the input field control in the upper right corner of the dialog window, next to the first text constant control (in the same way as described above for the text constant control).

17 Create two more input field controls (by duplicating the first, as above). These input field controls should have a height of 36. Align them horizontally with respect to each other and vertically with respect to the three text constant controls (as shown below).

18 From the **Insert** menu, choose **Push Button**.

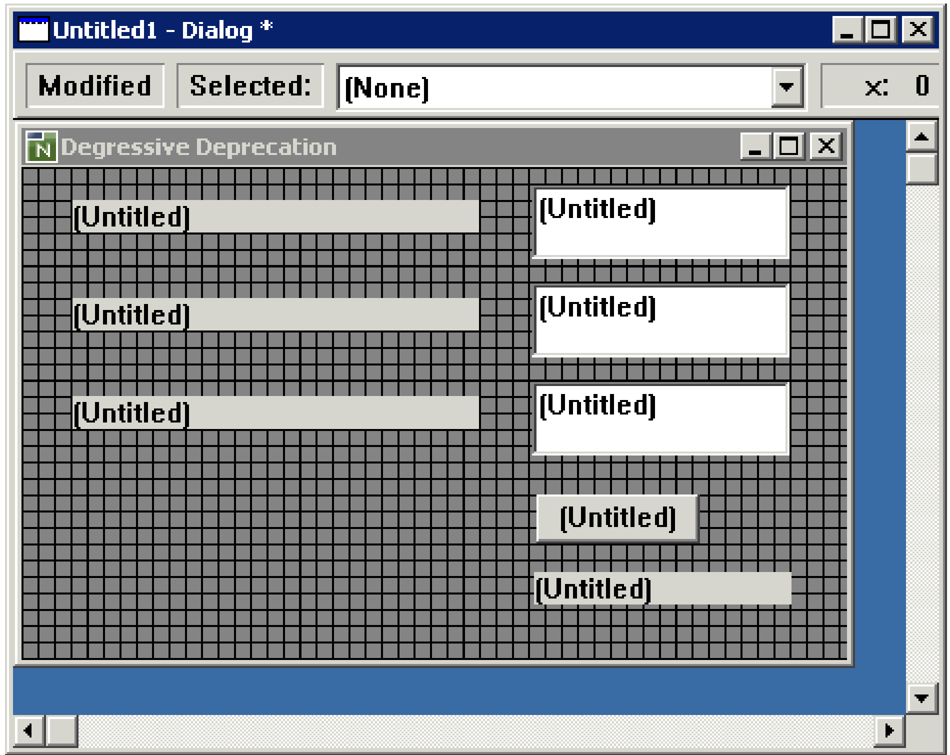
Oder:

Choose the toolbar button representing an push button control.

19 Position the push button control below the three input field controls.

20 Create a text constant control below the push button control.

Your dialog should now look like this:



Assigning Attributes to the Dialog Elements

To assign attributes to the dialog elements

1 Select the first text constant control `#TC-1` and from the **Control** menu, choose **Attributes**.

Oder:

Double-click the first text constant control `#TC-1`.

The corresponding attributes dialog box appears.

2 In the **String** text box, type in the text string to be displayed: "Initial Value".

3 Choose the **OK** button.

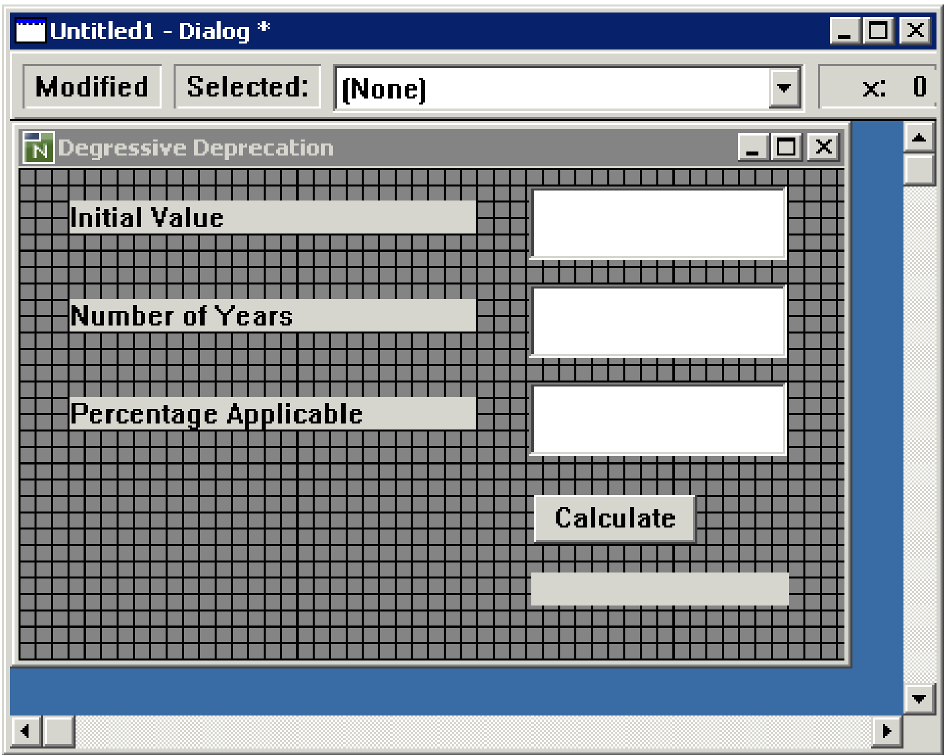
The attributes dialog box closes.

4 Set the following text strings for the two text constant controls below: "Number of Years" for `#TC-2` and "Percentage Applicable" for `#TC-3`.

5 From all three input field controls and from the fourth text constant control, remove any text strings (that is, the "Untitled" strings).

6 Set the following text string for the push button control: "Calculate".

Your dialog should now look like this:



Creating the Application's Local Data Area

The local data area in this application defines the application's linked variables. These linked variables receive the numeric values that the end user has entered in the input field controls. The variables and their values are used in the calculation of the push button control's click event handler code.

To prepare the creation of your local data area, your input field controls must use linked variables

1 Select the first input field control `#IF-1` and from the **Control** menu, choose **Attributes**.

Oder:

Double-click the first input field control `#IF-1`.

The corresponding attributes dialog box appears.

2 Choose the browse button (that is: the button with the three dots) to the right of the **String** text box.

The **Source for #IF-1.STRING** dialog box appears.

3 Select (and enable) the **Linked variable** option button.

4 In the **Variable name** text box, enter: "#INITIAL-VALUE".

5 Choose the **OK** button to close the **Source for #IF-1.STRING** dialog box and then choose the **OK** button to close the attributes dialog box.

6 Set the following linked variable names for the remaining two input field controls: "#YEAR-NUM" for `#IF-2` and "#PERC-APPLIC" for `#IF-3`.

To create the application's local data area

1 From the **Dialog** menu, choose **Local Data Area**.

The **Dialog Local Data Area** dialog box appears.

2 Define your local data as follows:

```
1 #INITIAL-VALUE (N6.2)
1 #PERC-APPLIC (N2.1)
1 #YEAR-NUM (N2)
```

3 Choose the **OK** button.

Natural will now be able to process the input data.

Attaching Event Handler Code to the Dialog Element

To attach event handler code

1 Select the push button control labelled **Calculate**.

2 From the **Control** menu, select **Event Handlers**.

A dialog box for the corresponding event handler definition section appears.

The `CLICK` event is preselected: when the end user clicks on this push button control, the specified Natural code will be triggered.

3 In the event handler editing area, enter the following Natural code in free form:

```
#RESULT:= #INITIAL-VALUE * ( ( ( 100 - #PERC-APPLIC )
/ 100 ) ** #YEAR-NUM )
MOVE EDITED #RESULT (EM=Z(5)9.99) TO #TC-4.STRING
```

4 Choose the **OK** button to close the dialog box.

Checking, Stowing and Running the Application

To check the application for syntax errors

1 From the **Object** menu, choose **Check**.

A dialog box comes up with a Natural error: a variable needs to be declared.

2 In the dialog box, choose the **Edit** button.

The dialog's code is displayed, the cursor pointing to the error (`#RESULT`).

3 Choose the **Cancel** button.

4 From the **Dialog** menu, choose **Local Data Area**.

5 Add the following definition:

```
1 #RESULT (N6.2)
```

6 Choose the **OK** button.

7 Check your application again.

The information message box should now confirm that the check was successful.

To stow your application

1 From the **Object** menu, choose **Stow**.

The **Stow Dialog As** dialog box appears.

2 Enter the name "Degrdep".

3 From the **Libraries** drop-down list box, select the library where you want the dialog to be stowed.

4 Choose the **OK** button.

The information message box now confirms that the dialog was stowed successfully.

To run your application

- From the **Object** menu, choose **Run**.

67 Basic Terminology

- Attribute 586
- Base Dialog 586
- Control 587
- Dialog 587
- Dialog Box 587
- Dialog Editor 587
- Dialog Element 587
- Event 588
- Event Handler 588
- Handle 588
- Item 588
- MDI - Multiple Document Interface 588
- MDI Child Window 589
- MDI Frame Window 589
- Modal Window 589
- SDI - Single Document Interface 589
- Popup 589
- Window 589

Event-driven Natural uses the following basic terminology:

Attribute

A property of a dialog or a dialog element which can assume specific values.

Example: If the `HAS-STATUS-BAR` attribute is set to `TRUE` for a dialog, then the dialog contains a status bar.

The following operations may be made on attributes:

Operation	Result
Query	In event handler code, you can query an attribute's value at runtime. Example: `#L:= #DLG$WINDOW.HAS-STATUS-BAR`

Set	In event handler code, you can set an attribute to a value in the global attribute list before you create a dialog element dynamically. Example:
	`#PUSH.STYLE:= 'O'`
	`PROCESS GUI ACTION ADD WITH #W`
	`PUSHBUTTON #PUSH`

Modify	In event handler code, you can modify an attribute value of an existing dialog element at runtime. Example: `#PUSH.STYLE:= 'C'`

Base Dialog

This is the main dialog of an application. It is started from the command line or via the object list. When this dialog is closed, all other dialogs of the application are closed as well.

Control

A type of dialog element. Examples: edit area control, push button control, list box control.

Dialog

A Natural object similar to a map or a program that represents a window in an event-driven application, plus all event handlers and attributes directly attached to the window. It can be a window, a modal window, a dialog box, an MDI child window, and an MDI frame window. The window as such is identified by its handle, the whole dialog is represented by the value of the system variable `*DIALOG-ID`.

Dialog Box

A special kind of dialog that is exclusively processed in an application. While this dialog is active, all other dialogs of the application are disabled and do not accept any user input. If a dialog invokes a dialog box with an `OPEN DIALOG` statement, the dialog returns from the `OPEN DIALOG` statement only after the dialog box is closed. This allows the application to return parameters from the dialog box to the dialog.

Dialog Editor

The Natural editor with which you create and maintain dialogs.

Dialog Element

Dialog elements are (in most cases) graphical elements inside a window that enable the end user to interact with the event-driven application. After a dialog has been created, and its attributes have been set, the programmer places the dialog elements inside the window; usually, this comprises a menu control, possibly a toolbar, and other elements, such as push button controls and input field controls. There are two types of elements: controls and items.

Event

Occurs when a user interacts with a dialog element. An event may also be sent from within a piece of code (user-defined event). Example: a click event occurs when the user uses the mouse to click on a push button control for which a piece of click event handler code has been specified. The system variable `*EVENT` contains the event name.

Event Handler

Programming code that is connected with a dialog element, and is triggered when a particular type of event occurs.

Handle

Identifies a dialog element in code and is stored in handle variables. Example: `#PB-1`.

Item

A type of dialog element that is part of a control. Example: selection box item, which is part of a selection box control.

MDI - Multiple Document Interface

Allows an application to manage several different documents or several views of the same document within the main application window (MDI frame window). These views or documents are displayed in separate MDI child windows.

MDI Child Window

Displays a view of a document within the MDI frame window of an MDI application.

MDI Frame Window

The parent window to all other child (document) windows in an MDI application.

Modal Window

Similar to a dialog box, except that if a dialog invokes a modal window with an `OPEN DIALOG` statement, the dialog returns from the `OPEN DIALOG` statement immediately after the modal window has completed opening.

SDI - Single Document Interface

As opposed to MDI applications, SDI applications do not have an MDI frame window that contains the document windows. Only a single view of a single document is displayed.

Popup

A dialog with style "Popup" is modeless and can be moved anywhere on the desktop.

Window

The basic type of window.

68 Event-Driven Programming Techniques

- Introduction
- How To Open and Close Dialogs
- How To Edit a Dialog's Enhanced Source Code
- How Dialogs, Controls and Items Are Related Hierarchically
- How To Define Dialog Elements
- How To Manipulate Dialog Elements
- How To Create and Delete Dialog Elements Dynamically
- How To Enable and Disable Dialog Elements
- Defining and Using Context Menus
- Using the Clipboard and Drag and Drop
- System Variables
- Generated Variables
- Message Files and Variables as Sources of Attribute Values
- Triggering User-Defined Events
- Suppressing Events
- Menu Structures, Toolbars and the MDI
- Executing Standardized Procedures

- Linking Dialog Elements to Natural Variables
- Validating Input in a Dialog Element
- Storing and Retrieving Client Data for a Dialog Element
- Creating Dialog Elements on a Canvas Control
- Label Editing in Tree View and List View Controls
- Working with ActiveX Controls
- Working with Arrays of Dialog Elements
- Working with Control Boxes
- Working with Date and Time Picker (DTP) Controls
- Working with Dialog Bar Controls
- Working with Error Events
- Working with a Group of Radio-Button Controls
- Working with Image List Controls
- Working with List Box Controls and Selection Box Controls
- Working with List View Controls
- Working with Nested Controls
- Working with a Dynamic Information Line
- Working with Spin Controls
- Working with a Status Bar
- Working with Status Bar Controls
- Working with Tab Controls
- Working with Tree View Controls
- Working with Dynamic Information Line and Status Bar
- Adding a Maximize/Minimize/System Button
- Defining Color
- Adding Text in a Certain Font

- Adding Online Help
- Defining Mnemonic and Accelerator Keys
- Dynamic Data Exchange - DDE
- Object Linking and Embedding - OLE

69 Introduction

This documentation addresses the more experienced GUI programmer and describes essential programming techniques. There are two ways to program in the dialog editor:

- Use the dialog editor's menu bar and toolbar to create new dialogs or dialog elements and use the attributes window to assign attribute values to them. The dialog editor will internally generate the corresponding Natural code.
- Open an event-handler section or an inline-subroutine section and specify Natural code explicitly. This code will be added to the code that is generated internally. You can also enter parameter data areas, global data areas and local data areas in the corresponding definition sections.

You can view the current dialog's generated and specified code by choosing **Object > List** in the dialog editor's menu bar.

If you want a hands-on demonstration of how to program with the dialog editor, refer to the SYSEXEVT library. This library contains sample dialogs demonstrating basic functionality. Before accessing the sample dialogs, read the README file. Then execute the MENU dialog.

Anmerkung: Code written in the dialog editor must be in structured mode.

If you want to execute a Natural application using dialogs, you must use a dialog to start this application.

For further information on event-driven programming see ***Introduction to Event-Driven Programming***.

70 How To Open and Close Dialogs

- Opening a Dialog 598
- Operands 598
- Passing Parameters to the Dialog 599
- Permanence in Creating, Passing and Checking Data 600
- Processing Steps When Opening a Dialog 601
- Closing Dialogs 602
- Initializing Attribute Values 602

Opening a Dialog

An event-driven application is started by executing the base dialog. Events triggered by the end user will then typically cause other dialogs to be started. The application ends when the base dialog is closed.

To open a dialog from anywhere within an event-driven application

- Use the statement `OPEN DIALOG`.

This statement causes the dialog to be loaded and the processing on its opening to be performed.

Control over processing returns from the opened dialog except for dialogs with the style "Dialog Box". For those dialog styles, control returns only after the dialog has ended.

The parameters passed are accessible only during the processing on the opening of a dialog (before-open and after-open events), except for when the parameters are declared as `BY VALUE` in the parameter data area of the opened dialog or when the dialog has the style "Dialog Box".

To open a dialog from anywhere within an event-driven Natural application, the following syntax is used:

```
OPEN DIALOG operand1     [USING]     [PARENT] operand2
                       [ [GIVING]    [DIALOG-ID] operand3  ]
                       [               { operand4...         } ]
                       [  WITH         {                     } ]
                       [               { PARAMETERS-clause   } ]
```

Operands

Operand1 is the name of the dialog to be opened. If the *PARAMETERS-clause* is used, *operand1* must be a constant (the name of a cataloged dialog).

Operand2 is the handle name of the parent.

Operand3 is a unique dialog ID returned from the creation of the dialog. It must be defined with format/length I4.

Passing Parameters to the Dialog

When a dialog is opened, parameters may be passed to this dialog.

As *operand4* you specify the parameters that are passed to the dialog.

With the *PARAMETERS-clause*, parameters may be passed selectively:

```
PARAMETERS [parameter-name=operand 4] _ END-PARAMETERS
```

Anmerkung: You may only use the *PARAMETERS-clause* if *operand1* is an alphanumeric constant and if the dialog is cataloged.

Parameter-name is the name of the parameter as defined in the parameter data area section of the dialog.

To avoid format/length conflicts between operands and parameters passed, see the `BY VALUE` option of the `DEFINE DATA` statement in the *Statements* documentation.

When passing parameters only with *operand4*, a dialog may be opened as follows:

```
/* The following parameters are defined in the calling dialog's parameter
/* data area (not in the parameter data area of the dialog to be opened):
 1 #MYDIALOG-ID (I4)
 1 #MYPARM1 (A10)
/* Pass the operands #MYPARM1 and 'MYPARM2' to the parameters #DLG-PARM1 and
/* #DLG-PARM2 defined in the dialog to be opened:
  OPEN DIALOG 'MYDIALOG' USING
  #DLG$WINDOW GIVING
  #MYDIALOG-ID WITH
  #MYPARM1 'MYPARM2'
```

When passing parameters selectively with the *PARAMETERS-clause*, a dialog may be opened as shown in the following example:

```
/* The following parameters are defined in the calling dialog's parameter
/* data area (not in the parameter data area of the dialog to be opened):
 1 #MYDIALOG-ID (I4)
 1 #MYPARM1 (A10)
/* Pass the operands #MYPARM1 and 'MYPARM2' to the parameters #DLG-PARM1 and
/* #DLG-PARM2 defined in the dialog to be opened:
  OPEN DIALOG 'MYDIALOG' USING
  #DLG$WINDOW GIVING
  #MYDIALOG-ID WITH PARAMETERS
```

```
  #DLG-PARM1=#MYPARM1
  #DLG-PARM2='MYPARM2'
END-PARAMETERS
```

Permanence in Creating, Passing and Checking Data

The term „permanence“ is used in Natural to denote data defined in a base dialog's local data area whose existence is guaranteed throughout the whole lifetime of the dialog. Data defined in the global data area are not kept permanent because the global data area can be exchanged while the application is executed.

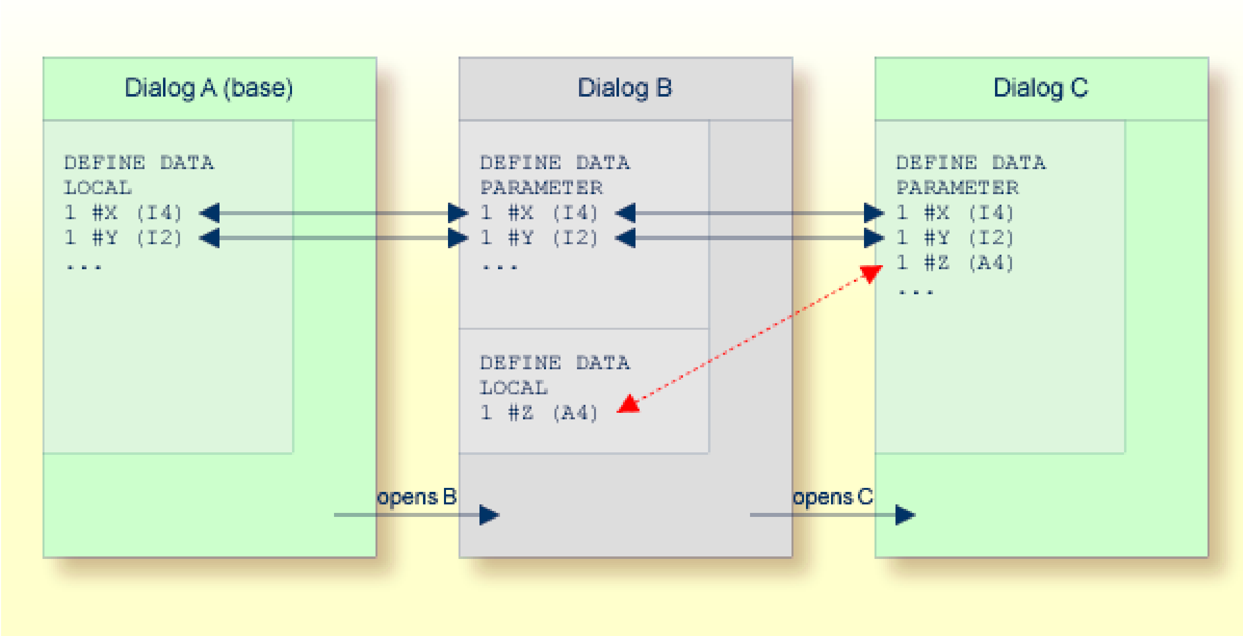
The reference to the permanent data is kept by saving the parameter data area internally during opening of the dialog. This reference is reused when

- a dialog element receives an event;
- all parameters passed from one dialog to another are permanent, provided they reference the base dialog's local data area.

Parameters are accessible

- during the before-open and after-open event processing on opening of a dialog or
- if *all of them* reference the base dialog's local data area.

The following example illustrates a case in which two parameters are kept permanently and one other is not. Assume the base dialog is dialog A. This base dialog now opens dialog B, passing parameters `#X` and `#Y`. After that, dialog B passes parameters `#X` and `#Y` on to dialog C. The `#X` and `#Y` parameters which are now in dialog C will be permanent, even if dialog B is closed. If, however, dialog B passes its own parameter `#Z` when opening dialog C, the parameter `#Z` is not permanent, because if dialog B is closed, the reference to its local data area is no longer valid. No parameter in dialog C is accessible (`#Z` does not reference the base dialog's local data area).



Processing Steps When Opening a Dialog

This section describes what happens when a dialog is opening. You can open a dialog either by executing it, for example from the command line, or by invoking it with an `OPEN DIALOG` statement.

- The dialog object is loaded and starts executing.
- The `BEFORE-ANY` event-handler section is executed, the value of the system variable `*EVENT` being `OPEN`.
- The `BEFORE-OPEN` event-handler section is executed.
- The dialog window is created as specified in the dialog editor.
- The `BEFORE-ANY` event-handler section is executed. `*EVENT` = `AFTER-OPEN`.
- All dialog elements are created as specified in the dialog editor.
- The dialog window and all dialogs are made visible except those that are `VISIBLE = FALSE`.
- The `AFTER-OPEN` event-handler section is executed.
- The `AFTER-ANY` event-handler section is executed. `*EVENT` = `AFTER-OPEN`.
- The `AFTER-ANY` event-handler section is executed. `*EVENT` = `OPEN` (not if the dialog's `STYLE` attribute value is "Dialog Box").

Closing Dialogs

To close a dialog dynamically, you specify the following:

```
CLOSE DIALOG  [USING] [DIALOG-ID] { operand1   }
                                  { *DIALOG-ID }
```

Operand1 is the identifier of the dialog as returned in the `OPEN DIALOG` statement.

Example:

```
CLOSE DIALOG *DIALOG-ID /* Close the current Dialog
```

The dialog will then be erased from the screen and removed from memory. All local data associated with the dialog will be gone.

Anmerkung: If a modal dialog is a child in a hierarchy of dialogs, the modal dialog should not close its parent(s) because this will result in a deadlock.

operand1

Operand1 is the name of the dialog to be closed.

To close the current dialog, you specify `*DIALOG-ID`.

Initializing Attribute Values

You can specify conditions for the opening and closing of a dialog: this applies to the before-open, after-open, and close events. These conditions can be used to initialize the attribute values in the dialog.

The following is an example of after-open event-handler code: Red foreground color is assigned to push buttons that the user must press after entering data in the associated input fields.

```
DEFINE DATA LOCAL
...
  1  #OK-BUTTON HANDLE OF PUSHBUTTON
  1 #CALC-BUTTON HANDLE OF PUSHBUTTON
  1 #SAVE-BUTTON HANDLE OF PUSHBUTTON
  1 #CONVERT-BUTTON
HANDLE OF PUSHBUTTON
...
END-DEFINE
...
#OK-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CALC-BUTTON.FOREGROUND-COLOUR-NAME := RED
#SAVE-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CONVERT-BUTTON.FOREGROUND-COLOUR-NAME := RED
```

If you want to modify attribute values of dialog elements and of the dialog before the dialog is opened (and displayed to the end user), do not specify this in the "before open" event-handler code, because the dialog elements and the dialog window are not yet created. Instead, create the dialog with the dialog editor and set the attribute `VISIBLE` to `FALSE` in the **Dialog Attributes** window. Then modify all the attribute values in the after-open event-handler code (when the handles are available). Then make the dialog visible with `VISIBLE = TRUE`.

Example:

```
DEFINE DATA LOCAL
...
 1 #DIA-1 HANDLE OF DIALOG
 1 #OK-BUTTON HANDLE OF PUSHBUTTON
 1 #CALC-BUTTON HANDLE OF PUSHBUTTON
...
END-DEFINE
...
/* AFTER OPEN event-handler code section
...
#OK-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CALC-BUTTON.FOREGROUND-COLOUR-NAME := RED
#DIA-1.VISIBLE := TRUE
```

71 How To Edit a Dialog's Enhanced Source Code

- What Is The Enhanced Source Code Format? ... 606
- Avoiding Incompatibilities Between Dialog Editor And Program Editor ... 607
- How To Use The Enhanced Source Code Format ... 608

What Is The Enhanced Source Code Format?

The enhanced source code format enables you to edit source code that has been generated by the dialog editor. You edit enhanced source code in a program editor window. When you edit a dialog, the dialog editor stores the results in internal structures. From these structures, source code is generated when you save, stow, list or execute any other system command on the dialog. Code is also generated when you refresh the program editor's source code window.

You can edit enhanced source code as you do any other Natural user code. The source code syntax is subject to a number of formal conventions, however. For a documentation of the enhanced source code syntax, see *Enhanced Source Code Format* in the *Dialog Editor* part of the *Editors* documentation.

When you execute a system command on a dialog you have just edited in the program editor source code window, the dialog editor updates its internal structures and refreshes the source code window.

Anmerkung: The dialog editor preserves code layout only in the user code sections, such as event handlers.

The dialog editor supports the following source formats:

- 213. This is the format generated by Natural Version 2.1.3 (New Dimension). It is supported for input only. You cannot generate 2.1.3 format with Natural Version 3.1 and Version 3.2.
- 22C. This is the format generated by Natural Version 2.2.2. In Natural for Windows and UNIX Version 4.1, dialogs can no longer be generated in this format. It, too, is supported for input only.
- 22D. This is the „enhanced“ source-code format that from now on is the standard. It is generated for compiling, storing, and editing dialogs in Natural Version 2.2.3 and above.

The characteristics of the enhanced source code format are:

- Dialog sources are readable and printable without requiring conversion.
- Dialog sources consist only of legal and fully documented Natural syntax.
- Dialog sources can be edited textually using program editor functions such as scanning for and replacing text.
- Dialog sources can be displayed in the Natural Debugger.
- Dialog sources are larger than 213 or 22C format sources (by a factor between 1.25 and 3.5).
- Any code that can be generated with the dialog editor can also be coded manually. For example, if you „draw“ a push-button control onto the user interface, the corresponding code is generated implicitly. You can also create this push-button control explicitly with the help of a source-code window that provides you with the functions of the program editor.

- You can switch between the dialog editor and the program editor by selecting the source code window or the dialog window. If you edit in either window, you need to synchronize your updates: (graphically) modifying the dialog locks the source code window and you may not make changes there. Correspondingly, if you change the source code, you may not make changes in the dialog window, which is locked. If your editor is locked, its status bar displays "Locked".

For dialogs in the old formats, this means:

- They remain unchanged until they are processed in the dialog editor. They can be compiled and executed in their old format.
- When you load them into the dialog editor, the dialogs are saved in the new format. If they are saved in the enhanced format, you must include the local data area NGULKEY1. Note that the storage size increases when the dialogs are saved.
- When you list or print them and you enable the "enhanced list mode" option, the dialogs are displayed using the enhanced source code format.

Avoiding Incompatibilities Between Dialog Editor And Program Editor

When you edit the enhanced source code format, note that some of the syntax elements accepted by the program editor are not accepted by the dialog editor. Enhanced source code editing is not intended as a new programming technique in addition to using the dialog editor:

- It may be syntactically acceptable to replace a dialog element's numeric coordinate (a `RECTANGLE-X` attribute value) with a variable reference. The dialog editor, however, will not accept this when the changes are synchronized, and will prompt you when you issue a command requiring the source code.
- The dialog editor may accept a reference to a variable's `STRING` attribute even if the variable is not declared, but the compiler will not accept this.

In the sections that are not user code, you should avoid such incompatibilities by adding only code that is acceptable to both the compiler and the dialog editor.

In the user code sections, such as in event-handler sections and in external or internal subroutines, your choice of programming techniques is not restricted by the dialog editor. In these sections, however, you have no visual editing support.

As a general rule, a mixed approach is often the best, especially when you use dialog-editor-generated code as a starting point.

Anmerkung: In the dialog editor, you can copy dialog elements to the clipboard and when you paste them into user code, they appear as text.

How To Use The Enhanced Source Code Format

To edit a dialog in the enhanced source code format

1. Load the dialog into the dialog editor.
2. From the **Dialog** menu, choose **Source Code**.

   Oder:

   Choose the "Source Code" toolbar button.

   Oder:

   Press CTRL+ALT+C.

   The dialog's source code window appears and the program editor is loaded. This editor enables you to scan for text strings, replace them, and so on. For more information on how to use the program editor, refer to *Program Editor*.

The enhanced source code format's syntactical conventions are documented in the section *Enhanced Source Code Format* of the *Dialog Editor* part of the *Editors* documentation.

Enhanced source code can be listed and printed as usual. You can also scan for strings by using the **Find** command of the **Edit** menu.

Anmerkung: If you are replacing strings with this command, this can make a dialog source incompatible with the dialog editor.

72 How Dialogs, Controls and Items Are Related Hierarchically

Dialogs and their dialog elements are organized hierarchically. Typically, the dialog window contains a number of controls. The controls are children of the window or of other controls which are capable of acting as containers. A control may contain a number of items. For example, a list box control may contain several list box items. The control is the parent of the items.

The dialogs themselves are also organized hierarchically. Every time the `OPEN DIALOG` statement is specified, the parent of the newly created dialog must be provided as a parameter. This parameter may be `NULL-HANDLE` or the handle of an existing dialog. If `NULL-HANDLE` is provided, the dialog belongs to the desktop rather than to any other dialog. This means that the dialog can be closed and minimized independently of any other dialog in the application. A dialog having an existing dialog as parent is closed or minimized when the parent dialog is closed or minimized.

The first dialog in an application plays a special role and is sometimes called the base dialog. When the base dialog is closed, all other dialogs in the application are also closed, whether they are children of the base dialog or not.

All children on one hierarchical level are sorted in the sequence of their creation. Each dialog element therefore always „knows“ its parent, its predecessor and successor (on the same hierarchical level), and its first and last child (if present). You can retrieve this information by using the following attributes:

- `PARENT`
- `PREDECESSOR`
- `SUCCESSOR`
- `FIRST-CHILD`
- `LAST-CHILD`

These attributes contain handle values of dialog elements. If their value is NULL, the dialog element has no parent, successor, or child. The following example demonstrates how to go through all dialog elements of a dialog.

Example 1:

```
1 #CONTROL HANDLE OF GUI

#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  ...
  #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
```

List box controls and list box items contain an additional attribute:

SELECTED-SUCCESSOR can be set for either the list box control itself or for any of its items. It points to the next selected item in a list box control. For the list box control itself, it points to the first selected item.

Example 2:

```
1 #ITEM HANDLE OF LISTBOXITEM

#ITEM := #LISTBOX.SELECTED-SUCCESSOR
REPEAT UNTIL #ITEM = NULL-HANDLE
  ...
  #ITEM := #ITEM.SELECTED-SUCCESSOR
END-REPEAT
```

The above example is the query necessary to find all selected items in a list box control where multiple selection is allowed (MULTI-SELECTION attribute).

73 How To Define Dialog Elements

- Introduction 612
- HANDLE OF GUI 613
- NULL-HANDLE 613

Introduction

Dialog elements are uniquely identified by a handle. A handle is a binary value that is returned when a dialog element is created. A handle must be defined in a `DEFINE DATA` statement of the dialog.

You can define a handle

- by creating a dialog or a dialog element with the dialog editor; in this case, the handle definition is generated;
- by explicitly entering the definition in a global, local, or parameter data area of the dialog;
- by explicitly entering the definition in a subprogram or a subroutine.

Anmerkung: Handles of ActiveX controls are defined in a slightly different way than the standard handle definition described below. This is described in ***Working with ActiveX Controls***.

A handle is defined inside a `DEFINE DATA` statement in the following way:

level handle-name [(*array-defintion*)] HANDLE OF *dialog-element-type*

Handles may be defined on any *level*.

Handle-name is the name to be assigned to the handle; the naming conventions for user-defined variables apply.

Dialog-element-type is the type of dialog element. Its possible values are the values of the `TYPE` attribute. It may not be redefined and not be contained in a redefinition of a group.

Examples:

```
1 #SAVEAS-MENUITEM HANDLE OF MENUITEM
1 #OK-BUTTON (1:10) HANDLE OF PUSHBUTTON
```

When you have defined a handle, you can use the *handle-name* with handle attribute operands in those Natural statements where an operand may be specified. With handle attribute operands, you can, for example, dynamically query, set, or modify attribute values for the defined *dialog-element-type*. This is the most important programming technique in the dialog editor. For details, see the section ***How To Manipulate Dialog Elements***.

If there is a dialog element handle of the same name in two different dialogs, the `PARENT` attribute ensures that Natural knows the difference between the two handles (two different `PARENT` values). Handles may be passed as parameters or may be assigned from one handle variable to another.

HANDLE OF GUI

In addition to the handle types referring to one dialog element, the generic handle type HANDLE OF GUI is available. In event-handler code, you can use HANDLE OF GUI to refer to the handle of any type of dialog element.

This can be useful, for example, if you are querying an attribute value in all dialog elements on one level: you go through the dialog elements one after the other; in the course of this query, it is not clear which type of dialog element is going to be queried next. Then a GUI handle makes it possible to query the next dialog element regardless of its type. This saves a lot of coding, because otherwise, you would have to query the attribute's value of each dialog element separately.

Example:

```
...
1 #CONTROL HANDLE OF GUI
...
#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  ...
  #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
```

NULL-HANDLE

The HANDLE constant NULL-HANDLE may be used to query, set or modify a NULL value of a HANDLE. Such a NULL value means that the dialog element is nonexistent (even if it has been created explicitly).

Example:

```
DEFINE DATA PARAMETER
  1 #PUSH HANDLE OF PUSHBUTTON
END-DEFINE
...
IF #PUSH = NULL-HANDLE
...
```

The HANDLE constant NULL-HANDLE represents the NULL value of a HANDLE variable or of an attribute with format HANDLE. For handle variables, the value indicates that the expression *handle.attribute* refers to the global attribute list. For attributes, this value indicates that no value is currently set.

74 How To Manipulate Dialog Elements

- Introduction 616
- Querying, Setting and Modifying Attribute Values 616
- Restrictions 617
- Numeric/Alphanumeric Assignment 618

Introduction

To manipulate dialog elements, Natural provides you with handle attribute operands. You use handle attribute operands wherever an operand may be specified in a Natural statement. This is the most important programming technique in event-handler code.

Wichtig: You must have **defined a handle**.

Anmerkung: ActiveX controls are manipulated in a slightly different way than the standard way described below. This is described in ***Working with ActiveX Controls***.

Handle attribute operands may be specified as follows:

```
handle.name-attribute.name[(index-specification)]
```

The *handle-name* is the handle of the *dialog-element-type* as defined in the HANDLE definition of the DEFINE DATA statement.

The *attribute-name* is the name of an attribute which has to be valid for the *dialog-element-type* of the handle.

Examples:

```
1 #PB-1 HANDLE OF PUSHBUTTON   /* #PB-1 is a handle-name of the
                               /* dialog-element-type PUSHBUTTON
RESET #PB-1.STRING...          /* #PB-1.STRING is the handle attribute operand
                               /* where STRING is a valid attribute-name of the
                               /* dialog-element-type PUSHBUTTON

1 #RB-1(1:5) HANDLE OF RADIOBUTTON /* #RB-1 is an array of five RADIOBUTTONs
IF #RB-1.CHECKED(3) = CHECKED      /* If the third radio-button control is
   THEN...                         /* checked ...
```

Querying, Setting and Modifying Attribute Values

In most applications, it will be necessary

- to set an attribute value before creating the dialog element,
- to modify the value after creating the dialog element, and
- to query an attribute value.

In some cases, it may be necessary to modify and query some attributes during processing, for example to query the checked/not checked state of a radio-button control or to disable (= modify) a menu item.

You can do that, for example, in the `ASSIGN`, `MOVE` or `CALLNAT` statements.

Examples:

```
1 #PB-1 HANDLE OF PUSHBUTTON      /* #PB-1 is a handle-name of the
...                               /* dialog-element-type PUSHBUTTON
#PB-1.STRING:= 'MY BUTTON'        /* Set or modify the value of the STRING
                                  /* attribute to 'MY BUTTON'
#TEXT:= #PB-1.STRING              /* Query the value of the STRING attribute
                                  /* and assign the value to #TEXT
CALLNAT 'SUBPGM1' #PB-1.STRING    /* Query the value of the STRING attribute
                                  /* and pass it on to the subprogram
```

When you use the *`handle-name`* variable only on the left side of the statements, as in the first of the three examples above, the attribute value is set or modified, that is, it is assigned the value of the specified *`operand`*.

When you use the *`handle-name`* variable on the right side of the statements, as in the second example, the attribute value is queried, that is, the value is assigned to the *`operand`*.

Once a handle has been defined (either explicitly in specified Natural code, or implicitly with the dialog editor), it can be used with most Natural statements. However, only a specific set of attributes can be queried, set or modified for a particular dialog element. To find out which values an attribute can have, see the section *Attributes* in the *Dialog Component Reference*.

Although an exact data type is specified for the values of most attributes, it is sufficient to supply move-compatible values to a handle attribute operand. The rules are the same as those for Natural variables.

Restrictions

Handle attribute operands must not be used in the following statements:

`AT BREAK`, `FIND`, `HISTOGRAM`, `INPUT`, `READ`, `READ WORK FILE`.

User-defined variables can be used instead.

Numeric/Alphanumeric Assignment

If you assign numeric operands to alphanumeric attributes, the values of these attributes will be in a non-displayable format. The Natural arithmetic assignment rules apply.

If you need a displayable format, you can use `MOVE EDITED`.

Examples:

```
#PB-1.STRING:= -12.34                         /* Non-displayable format
MOVE EDITED #I4 (EM = -Z(9)9) TO #PB-1.STRING /* Displayable format
```

The following edit masks may be used for the various format/length definitions of numeric operands:

Format/Length	Edit Mask
I1	-ZZ9
I2	-Z(5)9
I4	-Z(9)9
N*n.m*/P*n.m*	-Z(*n*).9(*m*)

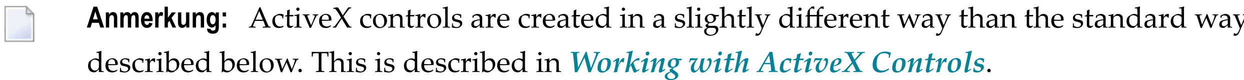
75 How To Create and Delete Dialog Elements Dynamically

- Introduction 620
- Global Attribute List 620
- Creating Dialog Elements Statically and Dynamically 620
- How to Handle Events of Dynamically Created Dialog Elements 622

Introduction

Dialog elements are usually added to a dialog by means of the dialog editor. However, they can also be created and deleted dynamically. This may be done, for example, when the layout of a dialog is strongly context-sensitive.

A dialog element is created dynamically with the `ADD` action of the `PROCESS GUI` statement. This action returns a handle to the newly created dialog element. As soon as the dialog element is created, this handle points to a set of attributes specified for the dialog element just created.



Anmerkung: ActiveX controls are created in a slightly different way than the standard way described below. This is described in *Working with ActiveX Controls*.

For more information on the actions available, and on the parameters that can be passed, see *Executing Standardized Procedures*.

Global Attribute List

By modifying any handle attribute operand of the form "*handlename.attributename*" (for example, `#PB-1.STRING`), you change an attribute value of the specific dialog element. As long as the dialog element is not yet created and the handle variable has its initial value (`NULL-HANDLE`), the handle attribute operand *handlename.attributename* refers to the global attribute list.

The global attribute list is a collection of all attributes defined for any dialog element. Natural contains one such collection. Whenever a dialog element is created, it „inherits" its attributes from this global attribute list. It does not inherit them when you create the dialog element with the `PROCESS GUI` statement action `ADD` using the `WITH PARAMETERS` option.

Creating Dialog Elements Statically and Dynamically

To define a dialog element statically (in the dialog editor), with an individual set of attributes, you must first set the attributes in the global attribute list to the desired values and then create the dialog element. After creation, the values of the attributes in the global attribute list remain intact. The next created dialog element gets the same attributes from the global attribute list as the previous one, except those that have been modified.

The status of the global attribute list as found in the „after open" event handler is influenced by the dialog elements defined statically. Therefore, before you start creating dialog elements dynamically in the „after open" event handler, you should reset the attributes by means of the `PROCESS GUI` action `RESET-ATTRIBUTES` to prevent your dialog elements from inheriting unexpected values

from the global attribute list. If you want to avoid this inheritance problem, use the PROCESS GUI statement action ADD with the WITH PARAMETERS option.

Unexpected values may also result from having attribute values that mean different things if used by different types of dialog elements. For example, the value "s" of the attribute STYLE means „scaled“ for the dialog element type bitmap control but „solid“ for the dialog element type line control.

The PROCESS GUI action ADD is used to define a dialog element dynamically. This clause of the PROCESS GUI statement enables you to specify the attribute values within the statement. The inheritance of attributes from the global attribute list does not affect the PROCESS GUI statement action ADD. The attributes specified in the statement are transferred to the global attribute list before the action ADD is performed.

Anmerkung: When you use the PROCESS GUI statement with Parameter Clause 2 of the ADD action, the global attribute list is not used or affected. For parameters which are needed to create the dialog element, but which were not specified in the WITH PARAMETERS section of the PROCESS GUI action ADD statement, the default value is taken. Apart from these, only the parameters which are passed explicitly in the parameter list are used to create the dialog element.

To create list-box and selection-box items dynamically, it may be more convenient to use the PROCESS GUI action ADD-ITEMS. This allows you to insert several items at a time.

Example:

```
/* #PB-A inherits the current settings of the global attribute list
#PB-A.STRING := 'TEST1'
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-A
#PB-B.STRING := 'TEST2'
/* #PB-B has the same attributes as #PB-A except STRING. This leads to #PB-B
/* covering #PB-A.
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-B
COMPUTE #PB-C.RECTANGLE-Y = #PB-B.RECTANGLE-Y + #PB-C.RECTANGLE-H + 20
/* #PB-B has the same attributes as #PB-A except RECTANGLE-Y
/* #PB-C will be located 20 pixels below #PB-B
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-C
```

To delete dialog elements dynamically, you use the PROCESS GUI action DELETE. You can also use this technique to delete dialog elements created with the dialog editor (at design time). You should, however, avoid using the handle of the deleted dialog element because this is invalid.

Dialog elements often do not have to be created dynamically. In some cases, it is sufficient to make dialog elements VISIBLE = TRUE and VISIBLE = FALSE, depending on the context. This technique is more efficient and easier to handle. It also enables you to „insert“ dialog elements anywhere in the navigation sequence.

Example:

```
DEFINE DATA LOCAL
   ...
   1 #PB-1 HANDLE OF PUSHBUTTON
   ...
END-DEFINE
...
#PB-1.VISIBLE := FALSE
...
IF...                        /* Logical condition
   #PB-1.VISIBLE := TRUE
END-IF
```

How to Handle Events of Dynamically Created Dialog Elements

When a dialog element is created dynamically, you cannot use the dialog editor to associate events to it. Instead, you must handle all events of all dynamically created dialog elements in the `DEFAULT` event. In this event, you must filter out which event occurred for which dialog element. The code for this is similar to the code generated by the dialog editor. The general structure is:

Example:

```
DECIDE ON FIRST *CONTROL
VALUE #PB-A
   DECIDE ON FIRST *EVENT
      VALUE 'CLICK'
         /* Click event-handler code
      NONE
         IGNORE
   END-DECIDE
VALUE #PB-B
   ...
VALUE #PB-C
   ...
END-DECIDE
```

In the case of event code for dynamically created ActiveX controls, *where event parameters are used*, it is necessary to precede the event code with an `OPTIONS 2` statement containing the name of the event, otherwise the compiler will not be able to process parameter references (e.g., `#OCX-1.<<PARAMETER-...>>`) successfully. However, in contrast to the implicit generation of the `OPTIONS` statement by the dialog editor for events for statically created controls, no `OPTIONS 3` statement should be coded in this case. Otherwise the dialog editor would falsely interpret the

`OPTIONS 3` statement as the end marker for the `DEFAULT` event, resulting in a scanning error on attempting to load the dialog.

Example:

```
DECIDE ON FIRST *CONTROL
VALUE #OCX-1  /* MS Calendar control
  DECIDE ON FIRST *EVENT
      VALUE '-602' /* DispID for KeyDown event
        OPTIONS 2 KeyDown
            /* KeyDown event-handler code containing parameter
            /* access (e.g. #OCX-1.<<parameter-shift>>)
        NONE
         IGNORE
 END-DECIDE
...
END-DECIDE
```

76 How To Enable and Disable Dialog Elements

During end-user interaction, it may be implicitly clear that certain dialog elements must not be used. For example, if a dialog requiring personnel data contains a group of radio button controls for marital status and an input field control for date of marriage, the input field control must be disabled whenever the marital status is other than "married".

There are two ways to do this:

- Use Natural code to enable/disable a dialog element dynamically.
- Use the dialog editor (to disable a dialog element initially).

The first method is used more often.

The Natural code might look like this:

```
/*First alternative
...
IF #RB-1.ENABLED = TRUE      /* Logical condition
   #IF-1.ENABLED := TRUE     /* Set ENABLED to TRUE
END-IF
...
/*Second alternative
#PB-1.ENABLED := #RB-1.ENABLED
```

When you use the dialog editor, you set the attribute `ENABLED` to `TRUE` by marking the **Enabled** entry in the dialog element's attributes window.

To disable editing in input-field controls, selection box controls and edit area controls, it is not always necessary to disable these dialog elements entirely. It may be sufficient to make them `MODIFIABLE = FALSE`.

77 Defining and Using Context Menus

- Introduction 628
- Construction 628
- Association 629
- Invocation 630
- Manual Invocation 633
- Sharing of Context Menus 635

Introduction

As from Natural v4.1.1, it is possible to create context menus for use within Natural applications. The context menus can be completely static (i.e., the menu contents are known in advance and can be built via the dialog editor) or wholly or partially dynamic (i.e., the menu contents and/or state depend on the runtime context and are not completely known at design time).

Construction

A context menu is very similar in concept to a submenu. Therefore, the same menu editor is used for editing a context menu as is used for editing a dialog's menu bar. Menu items can be added to context menus, and events associated with them, in exactly the same way as for menu-bar submenus. There are no functional differences to the menu bar editor, except that the **OLE** combo box (which is applicable only to top-level menu-bar submenus) will always be disabled. It should be noted, however, that any accelerators defined for context menu items will be globally available as long as that menu item exists. Furthermore, the accelerator will trigger the menu item for which it is defined even if the context menu is not being displayed or if the focus is on a control using a different context menu or no context menu at all.

The context menu editor may be invoked via either a new menu item, **Context menus...** on the **Dialog** menu, or via its associated accelerator (CTRL+ALT+X by default), or toolbar icon. However, because the context menu editor can only edit one context menu editor at a time, the context-menu editor is not invoked directly. Instead, the **Dialog Context Menus** window is shown, where operations on the context menu as a whole are made, and from which the menu editor for a given (selected) context menu can be invoked.

Internally, in order to distinguish between submenus and context menus, context menus have a new type, `CONTEXT MENU`. Otherwise, the generated code in both cases is identical. Here is some sample code illustrating the statements used to build up a simple context menu containing two menu items:

```
/* CREATE CONTEXT MENU ITSELF:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #CONTEXT-MENU-1
  TYPE = CONTEXTMENU
  PARENT = #DLG$WINDOW
END-PARAMETERS GIVING *ERROR
/* ADD FIRST MENU ITEM:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-1
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the first item'
```

```
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 1'
END-PARAMETERS GIVING *ERROR
/* ADD SECOND MENU ITEM:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-2
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the second item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 2'
END-PARAMETERS GIVING *ERROR
```

Note that if context menus or context-menu items are created dynamically in user-written code, the context menu or menu items will not be visible to the dialog editor. For example, the dynamically created menu item will not be visible in the context menu list box, and the dynamically created menu items will not be visible in the context menu editor.

Association

After creating a context menu, the context menu needs to be associated with a Natural object. Context menus are supported for almost all controls types capable of receiving the keyboard focus and for the dialog window itself. The full list includes ActiveX controls, bitmaps, canvasses, edit areas and input fields, list boxes, push buttons, radio buttons, scroll bars, selection boxes, table controls, toggle buttons, standard and MDI child windows, and MDI frame windows.

For all object types supporting context menus, the corresponding attribute dialogs in the dialog editor include a read-only combo box listing all context menus created by the dialog editor, together with an empty entry. The selection of the empty entry implies that no context menu is to be used for this object, and is the default.

Internally, the association is achieved by a new attribute, `CONTEXT MENU`, which should be set to the handle of a context menu. This attribute can be assigned at or after object creation time, and defaults to `NULL-HANDLE` if not specified, indicating the absence of a context menu. For context menus created by the dialog editor, the context menu is specified at control creation time as illustrated below:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
  HANDLE-VARIABLE = #LB-1
  TYPE = LISTBOX
  RECTANGLE-X = 585
  RECTANGLE-Y = 293
  RECTANGLE-W = 142
  RECTANGLE-H = 209
  MULTI-SELECTION = TRUE
```

```
  SORTED = FALSE
  PARENT = #DLG$WINDOW
  CONTEXT-MENU = #CONTEXT-MENU-1
  SUPPRESS-FILL-EVENT = SUPPRESSED
END-PARAMETERS GIVING *ERROR
```

The same syntax can also be used for controls created in user-written event code. In other cases, where the control was created by the dialog editor but the context menu was not, the context menu attribute must be assigned to the control after its creation, e.g., in the dialog's `AFTER-OPEN` event:

```
/* CONTEXT MENU SPECIFIED AFTER CREATION:

#LB-2.CONTEXT-MENU := #CONTEXT-MENU-2
```

Note that a context menu is not destroyed when an object using it is destroyed. Instead, it is destroyed when its parent object (typically, the dialog for which the context menu was defined) is destroyed. Similarly, the assignment of a new menu handle to the `CONTEXT MENU` attribute where one is already assigned does not result in the previous context menu being destroyed. Thus, using the above examples, neither of the following statements results in `CONTEXT-MENU-1` being destroyed:

```
PROCESS GUI ACTION DELETE WITH #LB-1            /* #CONTEXT-MENU-1 LIVES ON

#LB-1.CONTEXT-MENU := #CONTEXT-MENU-2           /* SAME HERE
```

Invocation

The context menu invocation process in Natural is as follows:

1. If the context menu is accessed via the mouse (i.e., secondary mouse button click), the target control is initially assumed to be the control immediately under the mouse cursor. Otherwise, if the context menu is accessed via the keyboard (i.e., either via the context menu key, if any, or via the key combination Shift+F10), the target control is initially assumed to be the control that currently has the keyboard focus.
2. The control's click position is set, relative to the target control's client area. If the context menu is accessed via the keyboard, the click position is set to (0, 0).
3. A `CONTEXT-MENU` event is raised for the target control, if not suppressed via the `SUPPRESS-CONTEXT-MENU-EVENT` attribute.
4. The target control's `CONTEXT-MENU` attribute is queried. Depending on its value and the type of the target control, the following action is taken:

- If the attribute is set to NULL-HANDLE and the target control is a dialog, the context menu invocation process is aborted, without any context menu having been displayed.
- If the attribute is set to NULL-HANDLE and the target control is a dialog element, the target control is assumed to be the dialog element's PARENT, and the context menu invocation process repeats starting with step 2 above.
- If the attribute is set to the handle of a context menu, this context menu is taken as being the context menu that needs to be displayed (i.e., the *target* context menu), and processing continues with step 5 below.

5. A BEFORE-OPEN event is raised for the target context menu, if not suppressed.
6. The target context menu's ENABLED attribute is queried. If it is set to FALSE, the context menu is not displayed.
7. Otherwise, a COMMAND-STATUS event is raised for the target dialog, if not suppressed. The target dialog is the dialog containing the target control, if it is a dialog element, or the target control itself, if it is a dialog.
8. The context menu is displayed at the click position set in step 2 above.

The actual navigation within the context menu and the triggering of the events associated with the menu items is done by Windows and Natural with no intervention from the application.

Note that the above process continues up through the control hierarchy, starting with the initial target control, until if finds a dialog or dialog element with a context menu (if any), and then uses that context menu.

The purpose of the CONTEXT-MENU event is to allow application to select the appropriate context menu (by modifying the target control's CONTEXT-MENU attribute) from a number of possible candidates according to the context. For an example of using multiple context menus, see ***Working with List View Controls***.

Similarly, the context menu's BEFORE-OPEN event gives the application the chance to modify the context menu according to the current program state. For example, menu items could be added or deleted, or particular menu items grayed or checked. Here is some sample code for the BEFORE-OPEN event:

```
/* DELETE FIRST MENU ITEM:
PROCESS GUI ACTION DELETE WITH #MITEM-1
/* CHECK SECOND MENU ITEM:
#MITEM-2.CHECKED := CHECKED
/* DISABLE THIRD MENU ITEM:
#MITEM-3.ENABLED := FALSE
/* INSERT NEW MENU ITEM BEFORE #MITEM-3:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-4
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the first item'
```

```
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 3'
  SUCCESSOR = #MITEM-3
END-PARAMETERS GIVING *ERROR
```

For context menus not created by the dialog editor, the handling of the `BEFORE-OPEN` event must be done in the `DEFAULT` event for the dialog. Note also that if a control or dialog is disabled, no context menu is displayed, and the `BEFORE-OPEN` event is also not triggered. The same applies if the context menu itself is disabled. For example:

```
#CONTEXT-MENU-1.ENABLED := FALSE        /* DISABLE CONTEXT MENU DISPLAY
```

Note that it is possible to disable the context menu in this way during the `BEFORE-OPEN` event, allowing selective disabling of the context menu depending on the mouse cursor position within the control. For example, it might be desired to only display a context menu if the mouse cursor is over a selected list-box item. Determining whether this is the case is possible via the use of two `PROCESS GUI ACTION` calls:

- `INQ-CLICKPOSITION` has been extended to controls other than bitmaps and canvasses to return the (X, Y) position of the right mouse button click within the control. In addition, these parameters are now optional, and a new optional parameter has been introduced that is set to `TRUE` if the context menu was accessed via the mouse, or `FALSE` if it was accessed by the keyboard. In the latter case, the click position is set to (0, 0). All this information is updated immediately prior to the sending of the `BEFORE-OPEN` event.
- `INQ-ITEM-BY-POSITION`. This allows translation of the relative co-ordinate returned by `INQ-CLICKPOSITION` applied to a list box to the corresponding item.

As an example of the use of these two new actions, consider the situation where we want to detect whether the cursor was over a selected list-box item when the right mouse button was pressed in order to determine whether to display a context menu or not. This can be achieved by the following code in the `BEFORE-OPEN` event of the associated context menu:

```
PROCESS GUI ACTION INQ-CLICKPOSITION WITH
   #LB-1 #X-OFFSET #Y-OFFSET
PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
   #LB-1 #X-OFFSET #Y-OFFSET #LBITEM
#MENU = *CONTROL
IF #LBITEM = NULL-HANDLE                /* NO ITEM UNDER (MOUSE) CURSOR */
  #MENU.ENABLED := FALSE
ELSE
  IF #LBITEM.SELECTED = FALSE           /* ITEM UNDER CURSOR DESELECTED */
     #MENU.ENABLED := FALSE
  ELSE                                  /* ITEM UNDER CURSOR IS SELECTED */
     #MENU.ENABLED := TRUE
```

```
  END-IF
END-IF
```

In some cases, it may be desired to automatically select the item under the mouse cursor if it is not already selected, clearing any existing selection. For list boxes, it is possible to achieve this by using the new `AUTOSELECT` attribute, either directly or via the new **Autoselect** check box in the **List Box Attributes** window in the dialog editor. If this attribute is set to `TRUE`, Natural will automatically update the selection before sending the `BEFORE-OPEN` event, if the context menu was invoked over an unselected list-box item.

For table controls, any change in the selection must be done via the application itself in the `BEFORE-OPEN` event. To make this possible, another new `PROCESS GUI ACTION` has been introduced for table controls:

- `TABLE-INQUIRE-CELL`. This returns the cell's row and column number (starting from 1) for a relative (X, Y) position within the table. This position can (and would typically be) the position returned by a previous call to `PROCESS GUI ACTION INQ-CLICKPOSITION`.

The `COMMAND-STATUS` event is an alternative location for the application to perform any updating such as graying and checking of commands (i.e., menu items, tool bar items and signals). If you are already using this event, you do not need to perform these actions in the `BEFORE-OPEN` event.

Manual Invocation

In addition to the automatic context menu invocation process described above, it is also possible to invoke a particular context menu manually at a specific position via the `SHOW-CONTEXT-MENU` action.

This is primarily intended for (but not restricted to) use with ActiveX controls where the automatic mechanism is not always applicable. This is because some ActiveX controls, depending on their internal implementation, do not raise the message used by Natural to trigger the context menu display. In such cases, if the ActiveX control raises an event when the secondary mouse button is pressed, the context menu can be manually displayed within the event handler for that event via this action.

For example, assuming we wish to display the context menu #CTXMENU-1 for the Microsoft Rich Textbox ActiveX Control, #OCX-1, we could use the following code in the control's MouseDown event handler:

```
IF #OCX-1.<<PARAMETER-Button>> = 2
  #X := #OCX-1.<<PARAMETER-x>> + 2
  #Y := #OCX-1.<<PARAMETER-y>> + 2
  PROCESS GUI ACTION SHOW-CONTEXT-MENU WITH
    #CTXMENU-1 #OCX-1 #X #Y GIVING *ERROR
END-IF
```

where the following local data definitions are assumed:

```
01 #X (I4)
01 #Y (I4)
```

Note that the above code first checks whether the secondary mouse button was pressed, then invokes a context menu manually, based on the position passed by the control. The position is, however, first corrected slightly to account for the fact that the position supplied by the control is relative to the ActiveX control (which has a 2-pixel sunken border), whereas the position used to display the context menu is assumed to be relative to the ActiveX control's *container* window (which has no border).

Note that some ActiveX controls may return coordinates in units other than pixels, such as twips (twentieths of a point). The following example shows how to convert co-ordinates (#X, #Y) from twips to pixels:

```
#CONTROL := *CONTROL
/* Convert x-coordinate
MULTIPLY #X BY #CONTROL.DPI
DIVIDE #X BY 1440
/* Convert y-coordinate
MULTIPLY #Y BY #CONTROL.DPI
DIVIDE #Y BY 1440
```

where #CONTROL is defined as HANDLE OF GUI, and #X and #Y are assumed to be of format I4.

The value 1440 is the number of twips per logical inch, whereas the DPI attribute applied to a dialog element returns the number of pixels per logical inch.

Sharing of Context Menus

It is of course possible to associate the same context menu with more than one object (i.e., control or dialog). For example:

```
#LB-1.CONTEXT-MENU := #CTXMENU-1
#LB-2.CONTEXT-MENU := #CTXMENU-1
```

In such a scenario, we need to be able to determine for which control the context menu was invoked. We cannot use `*CONTROL` in the `BEFORE-OPEN` event, because this will contain the handle of the context menu itself. Instead, it is necessary to inquire which control has the focus, since Natural automatically places the focus on the control for which the context menu is being invoked. Here is some sample `BEFORE-OPEN` event code illustrating the use of this technique:

```
PROCESS GUI ACTION GET-FOCUS WITH #CONTROL
DECIDE ON FIRST VALUE OF #CONTROL
  VALUE #LB-1
     #MITEM-17.ENABLED := FALSE
  VALUE #LB-2
     #MITEM-17.CHECKED := CHECKED
  NONE
     IGNORE
END-DECIDE
```

However, a better approach, which works in all cases, is to query the context menu's `CONTROL` attribute instead:

```
#CONTROL := *CONTROL
DECIDE ON FIRST VALUE OF #CONTROL.CONTROL
  ...
END-DECIDE
```

78 Using the Clipboard and Drag and Drop

- Introduction 638
- Clipboard Specifics 640
- Drag and Drop Specifics 641
- Drag and Drop Insertion Marks 643
- Drag-Drop Checklist 644

Introduction

Both clipboard and drag/drop data transfer make use of a logical clipboard at the Natural language level, allowing a single set of methods to handle both requirements. The `PROCESS GUI` actions for handling the logical clipboard are as follows: `OPEN-CLIPBOARD`, `SET-CLIPBOARD-DATA`, `CLOSE-CLIPBOARD`, `GET-CLIPBOARD-DATA` and `INQ-FORMAT-AVAILABLE`. Each Natural process has exactly one logical clipboard, which is why it is referred to in the product documentation as the „local“ clipboard.

`OPEN-CLIPBOARD` is the first step in building up the logical clipboard data. It takes an optional parameter (owner window), which is typically the handle of the control sourcing the data. If anything was previously on the logical clipboard, this action empties it. Note that you don't need to call this for drag and drop, because Natural does this implicitly before raising the `BEGIN-DRAG` event (see below).

`SET-CLIPBOARD-DATA` puts the actual data on the logical clipboard. The first parameter is the clipboard format, specified as a string. There are two pre-defined formats (defined in NGULKEY1 as CF-TEXT and CF-FILELIST), which are used for standard text transfer, and lists of files (suitable for data exchange with the Windows Explorer and many other applications) respectively. In addition, an arbitrary string (which should not begin with a dgit) should be used to indicate a private clipboard format that only Natural applications can understand (they just need to know the format string so they can pass it to `GET-CLIPBOARD-DATA` to retrieve the data). The second and subsequent arguments are an arbitrary number of data operands. These can be any combination of arrays (incl. index ranges) or scalars (incl. dynamic and large alpha variables). Arrays are internally expanded into their individual elements, which are then handled individually as for scalars.

For example, the following code:

```
DEFINE DATA LOCAL
1 #ARR(A1/2,3) INIT (1,1)<'A'> (1,2)<'B'> (1,3)<'C'>
                             (2,1)<'X'> (2,2)<'Y'> (2,3)<'Z'>
END-DEFINE
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT #ARR(*,*)
```

is equivalent to:

```
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT
  'A' 'B' 'C' 'X' 'Y' 'Z'
```

For the pre-defined formats, the operands must be alphanumeric (format A). For private formats, the data arguments can be of almost any type. Exception: handle variables (incl. `HANDLE OF OBJECT`) are not supported, because they are process-specific. The data for private formats is stored „as-is“ (i.e., no conversion).

Note that multiple data formats can be placed on the clipboard by performing a `SET-CLIPBOARD-DATA` action for each required format. However, any call to `SET-CLIPBOARD-DATA` for a particular format replaces any data that may already exist in that format.

Note also that the data is not placed on the Windows clipboard. This is done when the logical clipboard is closed (see below).

`CLOSE-CLIPBOARD` closes the logical clipboard, and places the data on the Windows clipboard, so that it becomes available for pasting into other applications. The data cannot be modified by `SET-CLIPBOARD-DATA` after this call. Note that this call is not necessary for drag and drop because you usually don't need to also make the dragged data available for pasting. Drag and drop can work directly with the logical clipboard.

`GET-CLIPBOARD-DATA` is used by the application performing the paste or acting as the drop target to retrieves the data from the drag-drop clipboard (if a drag and drop operation is in progress) or from the Windows clipboard otherwise. The drag-drop clipboard is a synonym for the logical clipboard belonging to the source Natural process. `SET-CLIPBOARD-DATA`, a clipboard format is specified, followed by an arbitrary list of data operands (with the same format type restrictions). For private formats, the operands need not have the same format type as used in the `GET-CLIPBOARD-DATA` action. For example, you can place an integer on the clipboard and read it back into a packed numeric (P) variable. Internally, a `MOVE` conversion is done. Therefore, if different format types are used for setting and getting the data, they must be `MOVE`-compatible.

For pre-defined formats, where the individual data items are either delimited by CR/LF (for CF-TEXT format) or by null-terminators (for CF-FILELIST format), only one item is usually read into each receiving field. Exception: If the last receiving field is a dynamic alpha variable, it receives all remaining data items, including the delimiters. This exception allows the application to use (for example) a single dynamic alpha variable to set and get multiple lines of data or multiple file/directory names. Regardless of the format used, if too many receiving operands are specified, the excess fields are reset (see the `RESET` statement). Note that individual data fields may be skipped by using the *`n`*`X` notation. For example, `5x` skips 5 data items (where a „data item“ is a single line for CF-TEXT format).

`INQ-FORMAT-AVAILABLE` is used for querying whether data is available in a given format (see specification for syntax). It is typically used to determine whether to enable or disable the **Paste** command, or whether to display the „no drop“ cursor for drag/drop operations.

Clipboard Specifics

The actual clipboard data transfer has been covered above. However, Natural allows you to define signals, menu items and toolbar items of the special types **Cut**, **Copy**, **Paste**, **Delete** and **Undo**, which (unlike normal commands) do not raise CLICK events. For input fields, edit areas, selection boxes and table controls, it's obvious what Natural should do, and Natural does this implicitly. For Natural, these commands now support list boxes and ActiveX controls. However, the mechanism is different in this case, because it is ambiguous as to how Natural should respond to these commands. Therefore, Natural needs some assistance from the application. This assistance comes in the form of six new events: CUT, COPY, PASTE, DELETE, UNDO and CLIPBOARD-STATUS, all of which are suppressible (via the new SUPPRESS-CUT-EVENT, SUPPRESS-COPY-EVENT, SUPPRESS-PASTE-EVENT, SUPPRESS-DELETE-EVENT, SUPPRESS-UNDO-EVENT and SUPPRESS-CLIPBOARD-STATUS-EVENT attributes). All six events are suppressed by default. The CUT, COPY, PASTE, DELETE and UNDO events are raised whenever the respective command is triggered. The corresponding event suppression flags are used by Natural to decide whether to enable or disable the corresponding command(s) in the user interface.

The CLIPBOARD-STATUS event is sent to the focus control during idle processing to give the application a chance to set these event suppression flags dynamically according to the context (e.g., whether or not there is an active selection). Natural raises this event before it queries the event suppression flags for the purpose of clipboard command status updating). Note that these new events are (currently) only sent to list boxes and ActiveX controls (and, of course, only if they currently have the focus). Input fields, selection boxes, etc., are still handled implicitly.

The CLIPBOARD-STATUS event is only raised if there is at least one clipboard command in the user interface that needs to be updated.

The following example shows a typical CLIPBOARD-STATUS event for a list box control:

```
DEFINE DATA LOCAL
1 #CONTROL HANDLE OF GUI
1 #FMT (A10) CONST<'MYDATAFMT'>
1 #AVAIL (L)
END-DEFINE
...
#CONTROL := *CONTROL
/*
/*
Cut, Copy & Delete are enabled if an item is selected,
/*or disabled otherwise
/*
IF #CONTROL.SELECTED-SUCCESSOR <> NULL-HANDLE
  #CONTROL.SUPPRESS-CUT-EVENT := NOT-SUPPRESSED
  #CONTROL.SUPPRESS-COPY-EVENT := NOT-SUPPRESSED
  #CONTROL.SUPPRESS-DELETE-EVENT := NOT-SUPPRESSEDELSE
```

```
  #CONTROL.SUPPRESS-CUT-EVENT := SUPPRESSED
  #CONTROL.SUPPRESS-COPY-EVENT := SUPPRESSED
  #CONTROL.SUPPRESS-DELETE-EVENT := SUPPRESSED
END-IF
/*
/* Paste command is enabled if data is available in a
/* recognized format, or disabled otherwise
/*
PROCESS GUI ACTION INQ-FORMAT-AVAILABLE
 WITH #FMT #AVAIL GIVING *ERROR
/*
IF #BOOL
  #CONTROL.SUPPRESS-PASTE-EVENT := NOT-SUPPRESSED
ELSE
  #CONTROL.SUPPRESS-PASTE-EVENT := SUPPRESSED
END-IF
```

Drag and Drop Specifics

Drag and drop operations can be triggered automatically (for list boxes and bitmap controls) or manually, via the new `PERFORM-DRAG-DROP` action (typically for ActiveX controls in response to control-specific mouse click or drag start events). For automatic drag/drop, the mouse cursor must be over the active selection (if any). For manual drag/drop, the parameters for `PERFORM-DRAG-DROP` include the handle of the control that should receive the drag/drop events (the drag source), and an optional flag indicating whether drag and drop should begin immediately, or only after the user moves the mouse a system-defined minimum number of pixels. Both automatic and manual drag/drop use the same code internally, so the same events are received in both cases.

Drag/drop is controlled by by two new I4 attributes, `DRAG-MODE` (for drag sources) and `DROP-MODE` (for drop targets). These attributes can be set to one of 8 values (defined in NGULKEY1): `DM-NONE` (no drag/drop allowed), `DM-COPY` (copy allowed), `DM-MOVE` (move allowed), `DM-COPYMOVE` (copy and move allowed), `DM-LINK` (link allowed), `DM-COPYLINK` (copy and link allowed), `DM-MOVELINK` (move and link allowed), `DM-COPYMOVELINK` (copy, move and link allowed). Link operations imply that the drop target should create a link to the source data, rather than creating a copy of it. For file operations, desktop shortcuts are typically used (not currently explicitly supported by Natural). Drag operations are only initiated if the source's `DRAG-MODE` attribute is set to something other that the default `DM-NONE` value. In addition, the application must respond to the `BEGIN-DRAG` event (see below).

Control types capable of acting as drop targets are: ActiveX controls, bitmap controls, list boxes, control boxes, edit areas, and dialogs (tab controls and table controls are planned for the future but are not currently supported). These windows are, however, only registered with OLE as drop targets if their `DROP-MODE` attribute is set to something other that the default `DM-NONE` value. During a drag/drop operation, OLE automatically searches up through the window hierarchy, starting with the window immediately under the cursor, until it finds a window that has been registered

as a drop target. This is the window that gets the OLE drop notifications and therefore is the window that receives the Natural drag/drop events (see below).

The new drag/drop related events are: `BEGIN-DRAG`, `END-DRAG`, `DRAG-ENTER`, `DRAG-OVER` and `DRAG-LEAVE`. In addition, the existing `DRAG-DROP` event (for the Mickey Mouse non-OLE drag/drop support for bitmap controls) is also used. All events are suppressible via the appropriate event suppression attributes (`SUPPRESS-BEGIN-DRAG-EVENT` etc.), all of which are `SUPPRESSED` by default.

The `BEGIN-DRAG` event (if not suppressed) is sent to the drag source on initiation of a drag operation. The application *must* use the `SET-CLIPBOARD-DATA` action to place some data on the drag/drop clipboard before returning from this event, otherwise the drag/drop operation is implicitly cancelled without the mouse cursor having changed. Note that there is no need to call either of the `OPEN-CLIPBOARD` or `CLOSE-CLIPBOARD` actions.

The `END-DRAG` event (if not suppressed) is sent to the drag source after a drag/drop operation has completed (even if the drag operation was cancelled). The main use of this event is to delete the source data if a Move operation occurred. The application can find out whether a Move operation has occurred by calling the existing `INQ-DRAG-DROP` action, which has been extended with two new optional integer output parameters. The first of these new parameters indicates which mouse buttons are currently pressed (1 = Left button, 2 = Right button, 4 = Middle button, or a combination thereof). The second new parameter is the one we need here, and contains the drop effect resulting from a drag/drop operation (`DM-NONE` if no drop or if the operation was cancelled, `DM-COPY` if a Copy operation was performed, `DM-MOVE` if a Move operation was performed, and `DM-LINK` if a link operation was performed.

The `DRAG-ENTER` event (if not suppressed) is sent to the drop target when the drag cursor (re-)enters the region occupied by the drop target. The application typically responds to this event by calling the `INQ-FORMAT-AVAILABLE` action to find out if a compatible data format is available on the clipboard, and then setting the `SUPPRESS-DRAG-DROP-EVENT` attribute accordingly. The `SUPPRESS-DRAG-DROP-EVENT` is important because it not only determines whether the `DRAG-DROP` event should be raised, but also informs Natural as to whether a drop should be allowed. After raising the `DRAG-ENTER` and `DRAG-OVER` events, Natural inspects the `SUPPRESS-DRAG-DROP-EVENT` attribute and displays a „no drop“ symbol. Otherwise, the drop effect is determined by the combination of the drag source's `DRAG-MODE` value, the drop target's `DROP-MODE` value, and the augmentation keys (SHIFT and CTRL) that are currently being pressed.

The `DRAG-OVER` event (if not suppressed) is frequently sent to the drop target as the drag cursor moves over the drop target. It can be used, for example, to update the drop emphasis (if any) as the user traverses the items within the control and/or to update the `SUPPRESS-DRAG-DROP-EVENT` attribute if the feasibility of a drop operation depends on the position within the drop target.

The `DRAG-LEAVE` event (if not suppressed) is sent to the drop target when the drag cursor leaves the region occupied by the drop target without a drop having occurred. This is mainly used (if at all) to remove the drop emphasis (if any) applied in the `DRAG-OVER` event.

The DRAG-DROP event (if not suppressed) is sent to the drop target when the user performs a drop. drag cursor leaves the region occupied by the drop target without a drop having occurred. The application should respond to this by effectively performing a Paste operation, using the current relative position within the control, if necessary. Both the relative position and the type of operation can be retrieved via the INQ-DRAG-DROP action. The latter is returned in the new (optional) „drop effect“ parameter (see the description of the END-DRAG event above for more information).

Drag and Drop Insertion Marks

For list boxes, a new "insertion mark (i)" style can be used to indicate that a dashed horizontal line be used to indicate the current insert position when the drag cursor is moved over the control (assuming it is a drop target). The application cannot query the insertion mark position directly, but can find out where to insert the data by querying the relative position within the control via the INQ-DRAG-DROP action, then passing these coordinates to the INQ-ITEM-BY-POSITION action, as in the following example:

```
DEFINE DATA LOCAL
1 #Y (I4)
1 #CONTROL HANDLE OF GUI
1 #ITEM    HANDLE OF GUIEND-DEFINE
...
/* DRAG-DROP event:
PROCESS GUI ACTION INQ-DRAG-DROP WITH 4X #Y GIVING *ERROR
*
IF #Y < 0
   #Y := 0
END-IF
#CONTROL := *CONTROL
PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
 #CONTROL 0 #Y #ITEM GIVING *ERROR
```

After the above code has executed, the variable #ITEM contains the handle of the item immediately following the insertion point. You can then dynamically insert one or more list box items at this position by calling the ADD action with the WITH PARAMETERS clause, setting the SUCCESSOR attribute to #ITEM.

Note that the correction for negative y-coordinate in the above example is necessary to cover the situation where the drop position is on the list boxes top border. If no correction would be made here, #ITEM would be set to NULL-HANDLE and the new list box item(s) would be added undesirably at the end of the list instead of at the beginning if we were to directly use #ITEM as the SUCCESSOR attribute, as described above.

Drag-Drop Checklist

For convenience, here is a brief overview of the steps involved in implementing drag-drop in Natural applications:

1. Set the `DRAG-MODE` for each drag source. If the drag source is a bitmap control, its `DRAGGABLE` attribute must also be set to `TRUE`.
2. Set `SET-CLIPBOARD-DATA` in the `BEGIN-DRAG` event for each drag source to provide the transfer data.
3. Set the `DROP-MODE` for each drop target.
4. In the `DRAG-ENTER` event, use the `INQ-FORMAT-AVAILABLE` action to set the `SUPPRESS-DRAG-DROP-EVENT` attribute to `NOT-SUPPRESSED (0)` if a supported clipboard format is available, or `SUPPRESSED (1)` otherwise. If the control can also act as a drag source and you need to prohibit drag-drop operations within the control, call `INQ-DRAG-DROP` to get the source control handle and compare it to the current control (`*CONTROL`), suppressing the drag-drop event if both are identical.
5. If the effect of the drop is position-sensitive within the target control, use the `INQ-DRAG-DROP` action within the `DRAG-OVER` event to get the current position, determine the item under the drag cursor (e.g. via the `INQ-ITEM-BY-POSITION` action) and set the `SUPPRESS-DRAG-DROP-EVENT` attribute appropriately. Highlight the current item if desired.
6. If the current item was highlighted in step 5 above, unhighlight it (if necessary) in the `DRAG-LEAVE` and (potentially) `DRAG-DROP` events.
7. Use `GET-CLIPBOARD-DATA` in the `DRAG-DROP` event to retrieve the transfer data and process it accordingly.
8. In the `END-DRAG` event for the drag source, delete the source data if the drop effect returned by `INQ-DRAG-DROP` is set to `DM-MOVE`.
9. If the drag source is an ActiveX control, call the `PERFORM-DRAG-DROP` action to initiate the drag-drop operation in response to a "MouseDown" event (for example) if a location within the current selection is clicked.

Example - Use of X-Arrays for Transferring Data

One of the problems in setting or retrieving data that may need to be placed or already have been placed on the Windows or drag-drop clipboard in response to a user interaction is being able to cope with an arbitrary amount of data at run-time. For example, the user may select a single, a few, or possibly even hundreds or thousands of list box items before performing a clipboard or drag-drop operation on them. With fixed-size arrays, one would have to define huge arrays to cope with the worst-case scenario, even though typically only a small percentage would be used most of the time.

There are two possible solutions to this problem available in Natural. The first way is to use a single dynamic alpha variable to contain all items to be set or retrieved. The application is then responsible for building up the items (including delimiters) in the dynamic variable before calling `SET-CLIPBOARD-DATA`, and for extracting the items from the dynamic variable after calling `GET-CLIPBOARD-DATA`. This approach is not possible for private formats, because these are not delimited.

The second approach is to make use of X-Arrays. For setting clipboard data, these behave similarly to fixed-size arrays, except that their size can be modified to contain exactly the number of elements needed in a specific situation. For example, if there are 17 items that need to be written to the clipboard, then you can use:

```
DEFINE DATA LOCAL
1 #X-ARR(A80/1:*)
1 #UPB (I4) INIT <17>
END-DEFINE
RESIZE ARRAY #X-ARR TO (1:#UPB)
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT #X-ARR(*)
```

instead of having to use a wastefully large fixed array, of which only a small range is used:

```
DEFINE DATA LOCAL
1 #ARR(A80/10000)
1 #UPB (I4) INIT <17>
END-DEFINE
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT ARR(1:#UPB)
```

When retrieving clipboard data, X-Arrays are even more useful, because the application does not know in advance how many items are on the clipboard. Passing all array elements (10,000 in the above example) would be relatively slow, because all unused elements need to be reset.

However, if an X-Array is used instead, Natural automatically resizes the array to (1:N), where N is the minimum of the number of items (remaining) on the clipboard and the array's maximum upper bound (as defined in `DEFINE-DATA`, where "*" indicates the maximum possible value). Note that there are three restrictions on the use of X-Arrays in conjunction with `GET-CLIPBOARD-DATA`:

- The X-Array must be the last (or only) parameter.
- Only 1-dimensional X-Arrays are supported.
- The X-Arrays defined range must include the element 1.

Here is an example program illustrating the use of a dynamic X-Array for retrieving clipboard data, including the use of a second X-Array to store and display the data lengths:

```
DEFINE DATA LOCAL
1 #FMT (A10) CONST<'MYDATAFMT'>
1 #X-ARR (A/1:*) DYNAMIC
1 #X-LEN (I4/1:*)
1 #UPB (I4)
1 #I (I4)
END-DEFINE
PROCESS GUI ACTION OPEN-CLIPBOARD GIVING *ERROR
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH #FMT
  'MIKE' 'FRED' 'JIM' 'LULU' 'FRANK' 'JANA' 'ELIZABETH'
'TONY'
  GIVING *ERROR
PROCESS GUI ACTION CLOSE-CLIPBOARD GIVING *ERROR
PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH #FMT #X-ARR(*)
  GIVING *ERROR
#UPB := *UBOUND(#X-ARR)
RESIZE ARRAY #X-LEN TO (1:#UPB)
FOR #I 1 #UPB
  #X-LEN(#I) := *LENGTH(#X-ARR(#I))
END-FOR
DISPLAY #X-ARR(*) (AL=10) #X-LEN(*) / '*** END OF DATA ***'
END
```

79 System Variables

Whenever you specify an event to occur with a given dialog element, the dialog editor generates code containing the Natural system variables `*CONTROL`, `*DIALOG-ID` and `*EVENT`.

During the processing, `*CONTROL` contains the dialog element's handle, `*EVENT` contains the event name and `*DIALOG-ID` identifies an instance of a dialog.

You can reference these system variables whenever you enter Natural code within the dialog editor. If, for example, the end user clicks on a push button control and the event handler calls a shared subroutine, you can use these system variables as logical condition criteria to trigger the subroutine.

For further details on these system variables, see the *System Variables* documentation.

80 Generated Variables

- #DLG$PARENT 650
- #DLG$WINDOW 650

#DLG$PARENT

You use this generated variable of type "user" to work with MDI child windows, for example. When you create a dialog, Natural generates this variable in order to hold the handle of the parent dialog. In event-handler code, you can, for example, use this variable to open an MDI child dialog from another MDI child dialog, as shown below.

Anmerkung: You should not use names for user-defined variables that begin with #DLG$ to avoid conflicts with generated variables.

Example:

```
OPEN DIALOG 'MDICHILD' #DLG$PARENT #CHILD-ID
```

#DLG$WINDOW

You use this generated variable to dynamically set the attributes within a dialog. When you create a dialog, Natural generates this variable in order to hold the handle of the dialog window. #DLG$WINDOW is the default name of this variable; you may change it by overwriting the **Name** entry in the upper left of the dialog's attributes window. In event-handler code, you can, for example, use this variable to minimize the dialog window if certain logical condition criteria are met, as shown below.

#DLG$WINDOW represents the graphical user interface aspects of a dialog, while the *DIALOG-ID system variable represents the runtime aspects. *DIALOG-ID must be used in OPEN DIALOG, CLOSE DIALOG and SEND EVENT statements.

Anmerkung: You should not use names for user-defined variables that begin with #DLG$ to avoid conflicts with generated variables.

Example:

```
...
IF ...
   #DLG$WINDOW.MINIMIZED := TRUE
END-IF
...
```

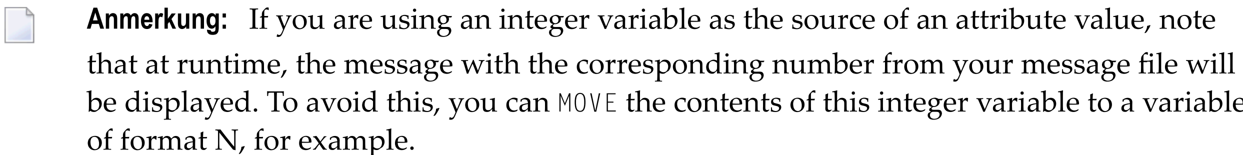
81 Message Files and Variables as Sources of Attribute Values

Most dialog elements have a `STRING` attribute. As an alternative to specifying the attribute value by typing in the text in the **String** entry of the attributes window, you can select a variable or a message file number from which the text is taken at runtime. In this case, the attribute value is determined by the variable's current value or the selected message file at the dialog element's creation time. You can also specify attribute sources for the `BITMAP-FILE-NAME`, `DIL-TEXT` and `ACCELERATOR` attributes.

To select a message file number or specify a variable

1. Invoke the dialog element's attribute window.
2. Choose the **Source** button to the right of the **String** entry.

   The **Attribute Source** dialog box appears. The default attribute source is "Constant"; you can also enter the number of the message file, or enter the variable name.



Anmerkung: If you are using an integer variable as the source of an attribute value, note that at runtime, the message with the corresponding number from your message file will be displayed. To avoid this, you can `MOVE` the contents of this integer variable to a variable of format N, for example.

82 Triggering User-Defined Events

- Introduction 654
- Passing Parameters to the Dialog 655

Introduction

Aside from standard events, such as before-open, you may define user-defined events for dialogs. User-defined events are useful whenever it is necessary for one dialog to cause an action to occur in another dialog.

A user-defined event occurs whenever you have specified a `SEND EVENT` statement in dialog A with the name of a user-defined event in the target dialog B. This target dialog B for which you wish to trigger the user-defined event must already be active. You can activate dialog B by using the `OPEN DIALOG` statement. If you do not issue the `OPEN DIALOG` statement first, the `SEND EVENT` statement will cause a runtime error.

You can define your own events for dialogs by choosing the **New** button in the **Events** dialog event handler menu or from the dialog's context menu. Enter any name for your newly-defined event and specify the corresponding event section. It is recommended that this name begin with "#" to distinguish your event from predefined events.

During execution of an event handler, the `SEND EVENT` statement triggers the user-defined event handler in a different dialog. After this user-defined event handler has been executed, control will be returned to the previous dialog, whose execution will resume at the statement following the `SEND EVENT` statement. This can be compared to a `CALLNAT` statement that causes a subprogram to be executed.

Similar to the `OPEN DIALOG` statement, parameters may be passed to the dialog. In order to pass parameters selectively (*PARAMETERS-clause*), you have to specify the name of the dialog in addition to the identifier of the dialog (*operand2*).

The `SEND EVENT` statement must not trigger an event in a dialog that is about to process an event. This is the case, for example, when dialog A sends an event to dialog B and the event handler in dialog B sends an event to dialog A which has not yet finished its event handling. A similar case is when dialog A opens dialog B and the before-open or after-open event contains a `SEND EVENT` back to dialog A.

To trigger a user-defined event, you specify the following syntax:

```
SEND EVENT operand1 TO [DIALOG-ID] operand2
                      [ { WITH operand3... USING [DIALOG] 'dialog-name' } ]
                      [ { WITH PARAMETRS-clause                          } ]
```

Operands

Operand1 is the name of the event to be sent.

`Operand2` is the identifier of the dialog receiving the user-defined event and must be defined with format/length I4. You can retrieve this identifier, for example, by querying the value of `#DLG$PARENT.CLIENT-DATA`.

Passing Parameters to the Dialog

It is possible to pass parameters to the dialog receiving the user event.

As *operand3* you specify the parameters which are passed to the dialog.

With the *PARAMETERS-clause*, parameters may be passed selectively.

PARAMETERS-clause

PARAMETERS [*parameter-name* = *operand3*]_ END-PARAMETERS

Anmerkung: You may only use the *PARAMETERS-clause* if the target dialog is cataloged.

Dialog-name is the name of the dialog receiving the user-defined event.

When you use only *operand3* to pass parameters, it might look like this:

```
/* The following parameters are defined in the dialog's
/* parameter data area:
1 #DLG-PARM1 (A10)
1 #DLG-PARM2 (A10)
1 #DLG-PARM3 (A10)
1 #DLG-PARM4 (A10)
/* When sending the user-defined event, pass the operands #MYPARM1 'MYPARM2' to
the parameters #DLG-PARM1 and #DLG-PARM2:
SEND EVENT 'MYEVENT' TO #DLG$DIA-ID WITH #MYPARM1 'MYPARM2'
```

When you use the *PARAMETERS-clause*, the user-defined event might look like this:

```
/* The following parameters are defined in the dialog's
/* parameter data area:
1 #DLG-PARM1 (A10)
1 #DLG-PARM2 (A10)
1 #DLG-PARM3 (A10)
1 #DLG-PARM4 (A10)
/* When sending the user-defined event, the operand #MYPARM2 is passed to the
/* parameter #DLG-PARM2 and the operand 'MYPARM3' is passed to the parameter
/* #DLG-PARM3:
SEND EVENT 'MYEVENT' TO #DLG$DIA-ID
```

```
USING DIALOG 'MYDIALOG'
WITH PARAMETERS
  #DLG-PARM3='MYPARM3'
  #DLG-PARM2=#MYPARM2
END-PARAMETERS
```

To avoid format/length conflicts between operands passed and their parameter definitions, see the `BY VALUE` option of the `DEFINE DATA` statement in the *Statements* documentation.

83 Suppressing Events

If an event occurs, normally an event handler will be triggered. It may, however, sometimes be necessary to dynamically suppress the execution of the event-handler code whenever the event has occurred. For example, if you want to modify the string of an input field control within the change-event handler, you must suppress the change event before modification to avoid an infinite loop because the modification itself triggers a change event.

The event-handler code may look like this:

```
...
IF...                                            /* Logical condition criteria
   #IF-1.SUPPRESS-CHANGE-EVENT := SUPPRESSED     /* Suppress the event
END-IF
...
```

By default, the dialog editor generates code to suppress all events for which no event handler code has been entered. In the dialog editor, you can also suppress an event with the **Suppress** option in the **Events...** dialog box.

If you suppress an event, the before-any and after-any events are also suppressed for this event.

84 Menu Structures, Toolbars and the MDI

- Creating a Menu Structure 660
- Parent-Child Hierarchy in Menu Structures 662
- Creating a Toolbar 662
- Sharing Menu Structures, Toolbars and DILs (MDI Application) 663

Creating a Menu Structure

A menu structure consists of three types of dialog elements:

- menu-bar controls,
- menu items,
- submenu controls.

A menu structure has one menu-bar control consisting of several menu items. The menu bar with its items is displayed directly beneath the window's title bar. Each menu item may be simple or may represent a submenu control, which allows you to pull down several menu items grouped vertically. Therefore, submenu controls may contain items representing a submenu control one level lower. A submenu control becomes visible when the representing item in the menu-bar control or the parent submenu control is clicked upon.

There are two ways to create menu structures:

- Use the dialog editor; or
- use Natural code.

If you use the dialog editor

1. Check the **Menu Bar** entry in the dialog's attribute window. Choose **OK**. When you go back to the dialog, a dummy menu-bar control appears.
2. Double-click on the dummy menu-bar control, or from the Natural Menu, select **Dialog > Menu Bar**, or use CTRL+M. The **Dialog Menu Bar** dialog box appears. This dialog box is divided into three group frames: menu bar, selected submenu and selected menu item.
3. In the selected menu items group frame, use **New** to add a menu item behind the selected position, or at the beginning. Now use the selected menu-item group frame to modify attribute values or add event handlers to the new menu item.

Normal menu items have a click event whose code is executed when the end user clicks on the menu item.

Anmerkung: The `MENU-ITEM-TYPE` of the menu item can also be "Separator", in which case the item is no text item.

If you use Natural code

1. Create a Menu Bar with the `PARENT` attribute set to "NULL-HANDLE" or "*windowhandle*".
2. To create a simple menu item: the `PARENT` attribute must have the value "*menubarhandlename*".

3. To create a submenu control: the submenu control's `PARENT` attribute must have the value "NULL-HANDLE" or "*windowhandlename*". Then create a menu item with `PARENT` = "*menubar-handlename*" and `MENU-HANDLE` = "*submenuhandlename*".
4. Then associate the menu bar with a dialog window by updating the window's `MENU-HANDLE` attribute with the handle of the menu bar as set in the first step.
5. The event handling for the dynamically created menu items must be done in the default event handler, as described in the section ***How to Create and Delete Dialog Elements Dynamically***.

The `PARENT` attribute determines when the menu bar or the submenu control will be destroyed. When `PARENT` = "*windowhandlename*", the menu bar/the submenu control will be destroyed when the window is destroyed. This is the default setting, which is also used by the dialog editor. If `PARENT` = NULL-HANDLE, the menu bar/the submenu control will be destroyed only when the application is terminated.

If you define the menu structure's handles inside a global data area, you can share these definitions among several dialogs.

To build the above menu structure

1 Define the handles of the menu-bar control, the menu items, and the submenu control(s) as the user-defined variables in the handler of the applicable event.

2 Create the controls and items by assigning values to the attributes (`PARENT`, ...) and by executing the `PROCESS GUI` statement action `ADD`.

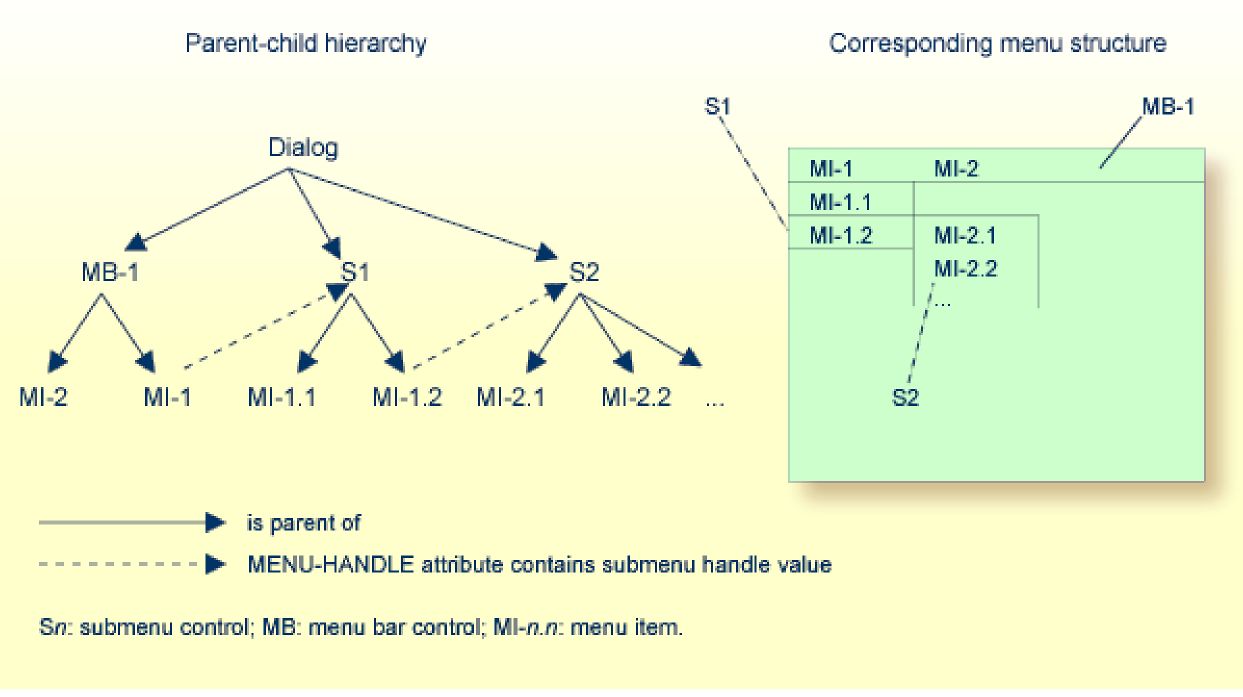
3 Create the controls and items in the sequence menu-bar control, submenu control with menu items.

4 Insert the controls and items in the sequence submenu control into menu-bar control, and menu-bar control into dialog window.

You can study how to build a menu structure in code by using the enhanced dialog list mode to list a dialog with an editor-built menu. To get a code model for creating a menu item, create a menu bar control with the dialog editor, go to the menu-bar control attributes window, cut a menu item and paste it into any chosen event-handler section. The generated code for the menu item appears.

Parent-Child Hierarchy in Menu Structures

Sometimes, it is necessary to use code for going through each element in a menu structure. For menus, the parent-child hierarchy is structured in a way that is not evident from the graphical representation of the menu structure.



In the above diagram, the first child of the dialog would be the menu-bar control. Its successor would be submenu control S1, and so on. To go from menu item MI-1 to submenu S1, you query for the `MENU-HANDLE` attribute value of MI-1. The value you get is the handle value of S1.

Creating a Toolbar

There are two ways of creating toolbars and their items:

- Use the dialog editor; or
- use Natural code to create them dynamically.

To use the dialog editor

1. Double-click on the toolbar or from the Natural Menu, select **Dialog > Toolbar**. The toolbar attributes window opens.
2. Add toolbar items by choosing the **New** button.
3. Assign bitmap file names and other attribute values to the new toolbar item.

If you want to use Natural code for dynamic creation, you can study how to build a tool bar in code. Use the enhanced dialog list mode to list a dialog with an editor-built tool bar.

Sharing Menu Structures, Toolbars and DILs (MDI Application)

An MDI (multiple document interface) application consists of a frame dialog that provides the menu structure, toolbar, and DIL shared among all child dialogs. An MDI frame dialog allows you to tile or cascade its child dialogs.

Anmerkung: You may only share the toolbar if the `PARENT` of the toolbar is the dialog of the highest level (the main dialog of an application).

To create an MDI frame dialog

1. Use the dialog editor, and go to the dialog object's attributes window.
2. Choose "MDI frame window" in the **Type** entry.

An MDI frame dialog must not contain dialog elements other than menu-bar control, submenu control, menu item, toolbar, and toolbar item.

To create an MDI child dialog

1. Use the dialog editor, and go to the dialog object's attributes window.
2. Choose "MDI child window" in the **Type** entry.

An MDI child dialog:

- can be moved and sized only inside the area of their MDI frame dialog;
- can be maximized to the full size of the area of their MDI frame dialog;
- can be minimized, after which its icon appears at the bottom of its MDI frame dialog;
- can have its own menu structure, toolbar, and DIL. Those do not appear inside the child dialog but are displayed in the MDI frame dialog when the child dialog is active. When another MDI child dialog becomes active, the menu structure, toolbar, and DIL change at the same time;

- can be arranged in a tile or cascade by setting a menu item's attribute `MENU-ITEM-TYPE` to the values "MDI Cascade" or "MDI Tile";
- can have its title added to the end of an MDI-WINDOWMENU type submenu control. By choosing one of these menu items, the corresponding MDI child dialog becomes active.

If you want to open an MDI child dialog from within an MDI frame dialog, you can, for example, create a menu item in a menu structure of an MDI frame dialog and define a click event for the menu item. You then write the `OPEN DIALOG` code for opening an MDI child dialog in the click event handler. The end user will open the MDI child dialog from within the MDI frame dialog by clicking on the menu item, triggering the click event handler.

Example:

```
OPEN DIALOG 'MDICHILD' #DLG$WINDOW #CHILD-ID
```

The first operand is the name of the dialog created by the dialog editor by selecting "MDI child window" in the **Type** selection box. The second operand is the parent of the new MDI child dialog. This must be the MDI frame dialog. The third operand is a Natural variable defined as I4 in the dialog's data areas. This variable receives the dialog ID returned by the system.

Anmerkung: `#DLG$WINDOW` is a generated variable.

You can also open an MDI child dialog from within another MDI child dialog (open a sibling of your MDI child dialog). Then you write a similar click-event handler as above:

```
OPEN DIALOG 'MDICHILD' #DLG$PARENT #CHILD-ID
```

The first and the third operands are the same as above. The second operand must be the parent of both MDI child dialogs.

Anmerkung: `#DLG$PARENT` is a generated variable.

85 Executing Standardized Procedures

- Introduction 666
- PROCESS GUI Statement 666

Introduction

For procedures frequently needed in event-driven applications, the following is available:

- a set of `PROCESS GUI` statement actions and
- a set of NGU-prefixed subprograms and dialogs in library SYSTEM.

Examples for frequently needed procedures are starting up a message box, reading the lines entered into an edit area control, or dynamically creating dialog elements.

For your convenience, the local data areas NGULKEY1 and NGULFCT1 are automatically included in the list of local data areas used by any new dialog.

- NGULFCT1 is necessary to use the NGU-prefixed subprograms and dialogs;
- NGULKEY1 lists reserved keywords to be used in any event-handler code. This enables you to refer to certain attribute values by the more meaningful keyword rather than by the numeric IDs. It also enables you to use meaningful dialog element names as parameters.

For more information on the `PROCESS GUI` statement actions, subprograms and dialogs available, and on the parameters that can be passed, refer to the *Dialog Component Reference*.

PROCESS GUI Statement

The `PROCESS GUI` statement is used to perform an action. An action in this context is a procedure frequently needed in event-driven applications.

As *action-name*, you specify the name of the action to be invoked.

As *operand1*, you specify the parameter(s) to be passed to the action. The parameters are passed in the sequence in which they are specified.

For the action `ADD`, you can also pass parameters by name (instead of position); to do so, you use the *PARAMETERS-clause*:

```
PARAMETERS [parameter-name = operand1]_ END-PARAMETERS
```

This clause can only be used for the action `ADD`, not for any other action.

As *operand2*, you can specify a field to receive the response code from the invoked action after the action has been performed.

86 Linking Dialog Elements to Natural Variables

In cases where you want to map database fields or other program variables to the user interface, input field controls and selection box controls may be linked to Natural variables. This makes it easier to modify and query them.

If the end user has entered data in an input-field control or a sebox control and sets the focus to another dialog element, a leave event occurs and the content (STRING) is moved to the variable. Thus, the variable is updated. Note that the variable will *not* be updated if the end user enters data and a change event occurs.

To refresh the content of the dialog element after the linked variable has been modified in code

- Use the PROCESS GUI statement action REFRESH-LINKS.

Modifying and querying input field controls with the ASSIGN statement would normally work like this:

```
...
#IF-1.STRING := '12345'
#TEXT := #IF-1.STRING
...
```

However, you can also link a Natural variable to the input field control or selection box control. You can also link an indexed variable to a dialog element or an array of dialog elements.

To link a variable in Natural code, set the attribute `LINKED` to `TRUE` and modify the attribute `VARIABLE` by setting it to the Natural variable name:

```
...
#IF-1.LINKED := TRUE
#IF-1.VARIABLE := MYVARIABLE
...
```

To use the dialog editor to enter the name of the Natural variable

1. Double-click on your input field control. The corresponding attributes window appears.
2. Choose the **Source** button to the right of the **String** entry. The **Source for** *`handlename`* dialog box appears.
3. Choose **Linked variable**.
4. Enter the variable name (such as `MYVARIABLE` in the example above).

There are two possibilities to link an indexed variable such as `MYVARIABLE (A20/1:5)`:

- you link a single dialog element to the indexed variable; then you specify the index, such as `MYVARIABLE(2)` in the variable name field of the **Source for** *`handlename`* dialog box, or
- you link an array of dialog elements to the indexed variable; then you do not specify an index in the variable name field. In this case, the occurrences of the array and the index of the variable must be compatible. `MYVARIABLE (A20/1:5)` could be linked to a one-dimensional array with up to five occurrences.

87 Validating Input in a Dialog Element

If an input field control or a selection box control is linked to a Natural variable, this dialog element may be checked automatically when it loses the focus to another dialog element in the same dialog. This enables you to validate the end user's input. An input field control or a selection box control will not be checked when the end user clicks on a menu item or switches to another application.

If you specify an edit mask with one of these two dialog elements, the field content is checked against this edit mask plus the Natural data type of the linked variable.

If no edit mask is specified, the field content is checked against the Natural data type only.

There are two ways of specifying an edit mask in an input field control or a selection box control:

- Use Natural code; or
- use the dialog editor.

The Natural code might look like this:

```
...
/* Create an input-field control
   1 #IF-1 HANDLE OF INPUTFIELD
...
/* Assign the Edit Mask
#IF-1.EDIT-MASK := '999'
```

To specify the edit mask with the dialog editor

- Open the input field control's attribute window and use the **Edit Mask** entry.

When the field check fails, a message box comes up where the end user can choose **Retry** or **Cancel**. **Retry** means that the entered text string remains unchanged and can be corrected. **Cancel** means that the field is reset to the current content of the linked variable.

88 Storing and Retrieving Client Data for a Dialog Element

- Introduction 672
- Integer Data 672
- Handle Data 673
- Keyed Alphanumeric Client Data 673
- Keyed Client Data in Native Format 676
- Key Enumeration 679

Introduction

This section refers to the association of arbitrary user-defined information („client data“) with a (dialog or) dialog element. There are various complementary ways of achieving this, which will be discussed in detail in the following sections. The attributes and actions relating to the manipulation of client data in Natural are (in the order they are discussed in this document):

- CLIENT-DATA attribute
- CLIENT-HANDLE attribute
- CLIENT-KEY attribute
- CLIENT-VALUE attribute
- SET-CLIENT-VALUE action
- GET-CLIENT-VALUE action
- ENUM-CLIENT-KEYS action

Integer Data

For a number of dialog element types, the CLIENT-DATA attribute may be used to associate a single arbitrary 4-byte integer value with the dialog element. This may be useful for linking data to a specific dialog element. A list box item, for example, can receive and pass on the ISN of a database record. The CLIENT-DATA attribute value may be changed at any time.

In Natural code, this might look like this:

```
DEFINE DATA
LOCAL
  1 #LBITEM-1 HANDLE OF LISTBOXITEM

  1 #ISN (I4)
   ...
END-DEFINE
...
READ...
   #LBITEM-1.CLIENT-DATA:= #ISN
END-READ
...
```

Anmerkung: The CLIENT-DATA attribute of a dialog is reserved for its dialog ID, and should not be used for the storage of user-defined client data.

Handle Data

Similarly, for all dialog element types, the `CLIENT-HANDLE` attribute may be used to associate a single arbitrary GUI object handle with the dialog element. For example, in the section ***Working with Dialog Bar Controls***, sample generic code is provided for building up a context menu containg entries for each tool bar control and dialog bar control in use by the dialog, allowing the user to individually show and hide them. In this example, the `CLIENT-HANDLE` attribute of each such menu item is set to the handle of the respective tool or dialog bar, allowing it to be both simply and directly retrieved when the menu item is clicked.

Keyed Alphanumeric Client Data

Anmerkung: The term „keyed“ refers to the ability to store multiple items of information for a given dialog element, each item being stored under a unique retrieval key.

Client data may also be set and retrieved as an alphanumeric string of up to 253 characters by using the `CLIENT-KEY` and `CLIENT-VALUE` attributes in combination.

To update a dialog element with a particular string

1. You first assign a value to the dialog element's `CLIENT-KEY` attribute, if this attribute does not already contained the desired value. This determines the key under which the string is to be stored for a dialog element.
2. You then assign an alphanumeric string to the `CLIENT-VALUE` attribute of the dialog element.

This enables you to store a number of key/value pairs for one dialog element.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
#LB-1.CLIENT-VALUE:= 'ANYSTRING'        /* The string to be stored
```

Anmerkung: In this and all following examples, the handle variable `#LB-1` is used, which (by convention) normally refers to a list box. However, with the exception of the `CLIENT-DATA` attribute, client data can be associated with GUI objects of any type, even those without a user interface, such as timers or signals.

To query a dialog element for a particular string

1. You first assign a `CLIENT-KEY` value to the dialog element, if this attribute does not already contained the desired value.
2. Then you query the `CLIENT-VALUE` attribute for the dialog element to retrieve the corresponding value.

If you query the `CLIENT-VALUE` of a `CLIENT-KEY` and there is no such key among the key/value pairs of the dialog element, an empty string (" ") is returned.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
IF #LB-1.CLIENT-VALUE EQ 'ANYSTRING' THEN
...
END-IF
```

If non-alphanumeric data is to be stored and retrieved, getting the data back into the original format may be a little more complicated, as shown below.

Example:

```
DEFINE DATA LOCAL
01 #DATE (D)
...
END-DEFINE

#LB-1.CLIENT-KEY := 'ANYKEY'
/* Store the current date
#LB-1.CLIENT-VALUE := *DATX

/* Retrieve it as a date (D) field
STACK TOP DATA #LB-1.CLIENT-VALUE
INPUT #DATE
```

The `STACK` statement retrieves the client value in alphanumeric form and places it one the Natural stack, from which the `INPUT` statement unstacks it into the specified variable, `#DATE`, implicitly converting the data from alphanumeric to date form. Alternatively, it would be possible to retrieve the client value into an alphanumeric variable, followed by explicitly converting it to the date field via a `MOVE EDITED` statement. However, the above approach has the advantage that it is not dependent on the date format (`DTFORM`), as well as not requiring the above-mentioned alphanumeric variable.

For some data types, such as dates and times, the default alphanumeric representation of the type (as used by the `CLIENT-VALUE` attribute) does not contain all the information contained in the ori-

ginal data type. For example, the default alphanumeric represention for time (T) values only contains the hours, minutes and seconds, and does not contain either the date component or tenths of a second. Similarly, the default alphanumeric represention for date (D) values does not contain century information. Thus, in order for the correct century to be assumed in the above example, it may be necessary to set the „Sliding Window“ (YSLW) parameter correctly before running the program.

If a dynamic alpha variable is used to directly receive the CLIENT-VALUE attribute value, the resulting value will have a length of 253 characters, being padded with blanks if necessary. This is due to the use of an attribute buffer of format A253 internally, and will be discussed later. The same effect is obtained when assigning an explicitly-defined A253 field to a dynamic variable. In either case, to prevent these trailing blanks from being stored in the dynamic variable, a COMPRESS statement should be used instead of a simple MOVE or assignment, as shown below.

```
DEFINE DATA LOCAL 01 #DYN (A) DYNAMIC ... END-DEFINE
#DYN := 'ANYSTRING' /* Set the client data #LB-1.CLIENT-KEY := 'ANYKEY'
#LB-1.CLIENT-VALUE
:= #DYN /* Retrieve value as 253-character string: #DYN := #LB-1.CLIENT-VALUE
/* Retrieve value without trailing blanks: COMPRESS #LB-1.CLIENT-VALUE INTO #DYN
```

Regardless of which of these approaches are used, any trailing blanks in dynamic alphanumeric variables are effectively lost if stored and retrieved via the CLIENT-VALUE attribute.

To delete a particular string for a dialog element

1. You first assign a CLIENT-KEY value to the dialog element, if this attribute does not already contained the desired value.
2. Then you RESET (or explicitly assign an all-blank value to) the CLIENT-VALUE attribute for the dialog element to delete the corresponding value.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY' RESET #LB-1.CLIENT-VALUE
```

Keyed Client Data in Native Format

As an alternative to setting client data in alphanumeric string form using the `CLIENT-KEY` and `CLIENT-VALUE` attributes in combination, the `SET-CLIENT-VALUE` and `GET-CLIENT-VALUE` actions may be used to store and retrieve client data directly in the supplied format, with no conversion. The value may, however, be stored in compressed form. In particular, trailing blanks in non-dynamic alphanumeric data are not stored, in order to save space. For example, if you supply an A253 field containing the value "FRED" followed by 249 filler blanks, only the A4 value "FRED" will be stored as client data internally. This latter optimization also applies to client data stored via the `CLIENT-VALUE` attribute.

The two techniques may be intermixed (i.e., one technique used to set the data and the other technique used to retrieve it). However, the use of the actions provides a number of advantages over the use of the attributes, as will become clear in the following sections.

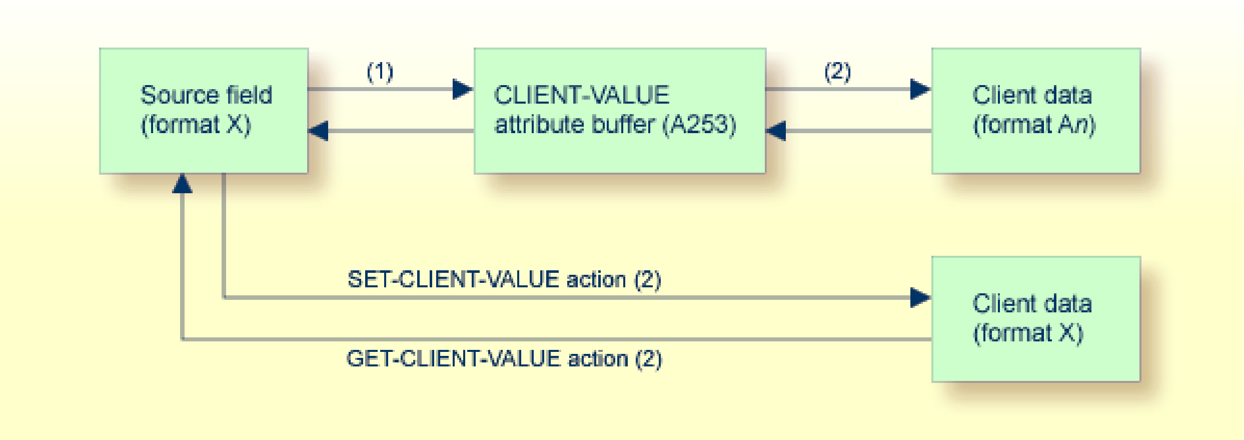
To update client data for a dialog element using the action-based technique

- Call the `SET-CLIENT-VALUE` action, passing the handle of the dialog element, the (client) key under which the value is to be stored, and the value itself. Alternatively, the key parameter can be omitted, in which case the current value of the dialog element's `CLIENT-KEY` attribute is implicitly used as the key.

Example:

```
#LB-1.CLIENT-KEY := 'ANYKEY' /* The following three statements are equivalent
ways of setting the same /* information: /* (1) attribute-based approach:
#LB-1.CLIENT-VALUE
:= 'ANYVALUE' /* (2) action-based approach, with explicitly-specified key PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 'ANYVALUE' 'ANYKEY' GIVING *ERROR /* (3)
action-based approach without key; CLIENT-KEY attribute implicitly used PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 'ANYVALUE' GIVING *ERROR
```

A significant advantage of storing client data via the `SET-CLIENT-VALUE` action is that there is no intermediate conversion to alphanumeric (A253) format involved, as is the case if the `CLIENT-VALUE` attribute is used. This is shown in the following diagram, where format X is used to imply any particular data type, and format A*n* represents an alphanumeric value stripped of any trailing blanks:



Here we see that the storage and retrieval of client data via the CLIENT-VALUE attribute is a two-step process (as is indeed the case for all attributes in Natural) depicted by the arrows "(1)" and "(2)" above, involving an attribute buffer corresponding to the defined format for the attribute - in this case A253. In contrast, the use of the SET-CLIENT-VALUE and GET-CLIENT-VALUE actions is a single step process that is effectively equivalent to performing step "(2)" alone, by-passing the conversion between the attribute buffer and the source or target field. This offers the following advantages (aside from being somewhat faster):

- Alphanumeric data longer than 253 characters may be stored without truncation, due to not having to pass through the attribute buffer.
- Handle values may be stored. These are incompatible with the use of an alphanumeric attribute buffer, because conversions between handles and alphanumeric fields are not allowed.
- If the data is being sourced from a dynamic alphanumeric variable, any trailing blanks are preserved. If the attribute buffer is used, trailing blanks become indistinguishable from (and are assumed to be) buffer filler characters and are thus stripped from the value when it is stored.
- Because the data is stored without conversion to and from alphanumeric format, non-alphanumeric data may be stored without any loss of information. For example, date information and tenths of a second are not lost when time values are stored, and century information is not lost when dates are stored.

In addition, there are other advantages to using the action-based approach for client data storage:

- Alphanumeric values consisting entirely of blanks may be stored. This is not possible via the CLIENT-VALUE attribute, as this would imply a delete operation.
- Error codes (e.g., in the case where an invalid control handle is passed) are returned in the GIVING field (if specified), without standard error handling necessarily being invoked (although this can be achieved, if desired, by the use of GIVING *ERROR).

To query client data for a dialog element using the action-based technique

- Call the `GET-CLIENT-VALUE` action, passing the handle of the dialog element, the (client) key for which the value is to be retrieved, and a field to receive the value itself. Alternatively, the key parameter can be omitted, in which case the current value of the dialog element's `CLIENT-KEY` attribute is implicitly used as the key.

Example:

```
DEFINE DATA LOCAL 01 #VALUE (A253) ... END-DEFINE PROCESS GUI ACTION GET-CLIENT-VALUE
WITH #LB-1 #VALUE 'ANYKEY' GIVING *ERROR IF #VALUE <> ' ' /* Value found ... ELSE
/* Value not found ... END-IF
```

Note that the format of the field specified to receive the value must be `MOVE`-compatible with the format of the stored value.

If the specified key is not found for the specified dialog element, the value field is `RESET`. For example, an alphanumeric receiving field is filled with blanks, and a numeric receiving field is set to zero.

However, if such values can be explicitly stored for this key by the program, the value alone cannot be used to determine whether the requested client data was found.

To query client data if resetted values are being explicitly stored

- Call the `GET-CLIENT-VALUE` action, also passing (in addition to the standard parameters mentioned above) a field of type L to receive the found/not found status.

Example:

```
DEFINE DATA LOCAL 01 #VALUE (A253) 01 #FOUND (L) ... END-DEFINE * PROCESS GUI
ACTION GET-CLIENT-VALUE WITH #LB-1 #VALUE 'ANYKEY' #FOUND GIVING *ERROR * IF #FOUND
... END-IF
```

The main advantage of reading client data via the `GET-CLIENT-VALUE` action is again the avoidance of going via an attribute buffer (see earlier diagram), implying that no intermediate conversion to or from alphanumeric (A253) format involved, as is the case if the `CLIENT-VALUE` attribute is used. Instead, the stored data is converted directly to the format of the receiving field for the value. This offers the following advantages:

- Alphanumeric data longer than 253 characters may be retrieved, without being truncated to the length of the (not used) `CLIENT-VALUE` attribute buffer.

- Handle values may be retrieved, which are not MOVE-compatible with the alphanumeric format of the CLIENT-VALUE attribute buffer.
- If the data is being read into a dynamic alphanumeric variable, any trailing blanks in stored alphanumeric data are preserved. If the CLIENT-VALUE attribute is used, the dynamic variable would receive the buffer's filler characters and be unable to distinguish them from any trailing blanks in the original data.

In addition, Stored alphanumeric values consisting entirely of blanks may be recognized. This is not possible via the CLIENT-VALUE attribute, as there is no way to distinguish them from the implicit „not found“ value.

To delete client data for a dialog element using the action-based technique

- Call the SET-CLIENT-VALUE action, passing the handle of the dialog element and the (client) key for which the value is to be deleted, but omitting the value itself. Alternatively, the key parameter can be omitted, in which case the current value of the dialog element's CLIENT-KEY attribute is implicitly used as the key.

Example:

```
/* No value supplied => delete any existing value for specified key PROCESS GUI
ACTION SET-CLIENT-VALUE WITH #LB-1 1X 'ANYKEY' GIVING *ERROR /* Alternatively,
a mixed attribute/action approach can be used: #LB-1.CLIENT-KEY := 'ANYKEY' PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 GIVING *ERROR
```

Key Enumeration

The above sections have dealt with the creation, updating, querying and deletion of client key and client value data. In most cases this is enough. However, in some situations, the keys that are being used by a dialog element are either not known to the code that reads them, or it is necessary to be able to verify that the expected keys are present for debugging or testing purposes. The iterative process of retrieving the keys currently being used by a particular dialog element is known as *key enumeration*.

To enumerate the client keys for a dialog element

1. Call the ENUM-CLIENT-KEYS action, passing the handle of the dialog element for which the client keys should be enumerated, but omitting the key parameter. This has the effect of resetting the dialog element's enumeration cursor (i.e., position) back to the beginning of its internal key list. Since the enumeration cursor is initially reset when a dialog element is created, this step is strictly not required for the first key enumeration for a particular dialog

element. However, it is good practice to explicitly reset the cursor in this manner, in order to make the enumeration context-insensitive.

2 Call the `ENUM-CLIENT-KEYS` action again, passing the handle of the dialog element and the key parameter, into which the first key (if any) will be returned.

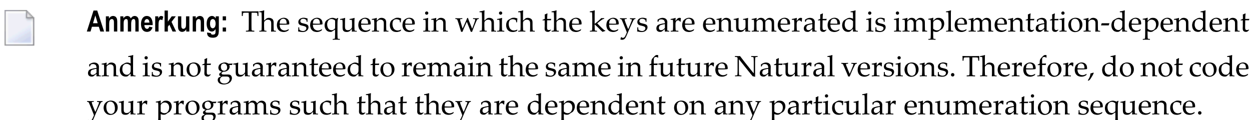
3 If the key field was internally `RESET` to blanks by the above call, this indicates that no (more) keys remain, and the program should terminate the enumeration process.

4 Otherwise, go back to step 2 in order to retrieve the next key (if any).

Example:

```
 /* Enumerate and output all client keys in use by
control #LB-1: /* (1) Reset enumeration cursor: PROCESS GUI ACTION ENUM-CLIENT-KEYS
WITH #LB-1 GIVING *ERROR /* (2) Enumerate and delete the keys one-by-one: REPEAT
PROCESS GUI ACTION ENUM-CLIENT-KEYS WITH #LB-1 #LB-1.CLIENT-KEY GIVING *ERROR
IF #LB-1.CLIENT-KEY <> ' ' RESET #LB-1.CLIENT-VALUE /* delete the key END-IF WHILE
#LB-1.CLIENT-KEY <> ' ' END-REPEAT
```

This example illustrates that the `ENUM-CLIENT-KEYS` action is tolerant of keys being deleted during the enumeration process. If (as shown here) the last enumerated (i.e., „current“) key is deleted, Natural automatically moves the internal enumeration cursor to its predecessor in th enumeration sequence, or resets it if there no predecessor. In either case, the next key returned by `ENUM-CLIENT-KEYS` is the one that would have been returned had the previous key not been deleted.



Anmerkung: The sequence in which the keys are enumerated is implementation-dependent and is not guaranteed to remain the same in future Natural versions. Therefore, do not code your programs such that they are dependent on any particular enumeration sequence.

89 Creating Dialog Elements on a Canvas Control

You can use a canvas control as a background to draw the following dialog elements on it: the rectangle, line and graphictext controls. These dialog elements „visualize“ information. You can, for example, create three or four rectangle controls, fill them with color and change their size at runtime. This way, you can build your own bar chart.

Once you have created a canvas control in the dialog, you can go on to create the rectangle, line and graphictext controls in it.

Anmerkung: Graphictext controls do not repaint the background of the rectangle in which they are located. The background of the rectangle is specified at creation time of the graphictext control. What they do repaint is only the text specified in the text attribute.

To create dialog elements on a canvas control

- Use the `PROCESS GUI` statement action `ADD`.

The rectangle, line and graphictext controls are then displayed inside the borders of the canvas control; if they exceed the canvas borders, they are clipped.

The following attributes are useful for controlling the behavior of the canvas control and the dialog elements on it:

- `OFFSET-X` and `OFFSET-Y` determine the x and y axis offset of the canvas control's upper border against the upper border of the area by which the rectangle, line or graphictext control have exceeded the canvas control's borders.
- `RECTANGLE-X`, `RECTANGLE-Y`, `RECTANGLE-W` and `RECTANGLE-H` determine the size of a rectangle control and its position relative to the underlying canvas control.
- `P1-X`, `P1-Y`, `P2-X` and `P2-Y` determine the start position (P1*xx*) and the end position (P2*xx*) of a line control relative to the underlying canvas control.

The following example illustrates how to create a canvas control

```
/* In the dialog's local data area, the following must be defined:
01 #CNV1 HANDLE OF CANVAS
01 #XAX HANDLE OF LINE
01 #YAX HANDLE OF LINE
01 #H1 HANDLE OF RECTANGLE
01 #H2 HANDLE OF RECTANGLE
01 #H3 HANDLE OF RECTANGLE
01 #H4 HANDLE OF RECTANGLE
01 #RESPONSE (I4)
/* In the dialog's AFTER-OPEN event handler, the following must be defined:
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #DLG$WINDOW
  TYPE = CANVAS
  HANDLE-VARIABLE = #CNV1
  RECTANGLE-X = 20
  RECTANGLE-Y = 20
  RECTANGLE-W = 200
  RECTANGLE-H = 200
  STYLE = 'F'
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = LINE
  HANDLE-VARIABLE = #YAX
  STYLE = 'S'
  P1-X = 20
  P1-Y = 20
  P2-X = 20
  P2-Y = 180
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = LINE
  HANDLE-VARIABLE = #XAX
  P1-X = 180
  P1-Y = 180
  P2-X = 20
  P2-Y = 180
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
```

```
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H1
  RECTANGLE-X = 20
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -60
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = RED
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H2
  RECTANGLE-X = 40
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -40
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = BLUE
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H3
  RECTANGLE-X = 60
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -55
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = GREEN
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H4
  RECTANGLE-X = 80
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -80
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = MAGENTA
END-PARAMETERS
GIVING RESPONSE
```

90 Label Editing in Tree View and List View Controls

- Introduction 686
- Label Editing 686
- Changing an Item's Label Programmatically 688

Introduction

This section describes the process of editing item labels for both tree view and list view controls. The word „item“ is therefore used throughout, in place of „tree view item“ and „list view item“, respectively.

Label Editing

The editing of an item's label, unless prohibited (see below), may be initiated in one of three ways:

1. By the user, by clicking on the label of a selected item.
2. By the user, by pressing the F2 key (whereupon the item with the focus rectangle, if any, is edited).
3. By the program, by calling the EDIT-LABEL action.

Regardless of the means of initiation, the sequence of actions taken by Natural in response is identical:

1. The control's MODIFIABLE attribute is examined. If this is FALSE (e.g., the **Modifiable** option was not checked in the control's attributes window), no further action occurs and label editing mode is not entered.
2. The control's ITEM attribute is set to the handle of the item for which label editing was requested (the „target“ item).
3. Unless suppressed, a BEFORE-EDIT event is raised for the control.
4. The target item's MODIFIABLE attribute is examined. If this is FALSE, no further action occurs and label editing mode is not entered.
5. Label editing mode is entered. The user may cancel any changes he has made via the ESC key. In this case, the original label is restored, edit mode exited, and no further action taken. Alternatively, the user can commit the changes (e.g. by pressing the ENTER key or setting the focus to another window or control).
6. The target item's STRING attribute is updated with the new label text.
7. Unless suppressed, an AFTER-EDIT event is raised for the control.
8. If the item's label is no longer identical to the item's STRING attribute (i.e., the application modified the attribute during the AFTER-EDIT event), the item's label is updated accordingly.

The purpose of the BEFORE-EDIT event is twofold. Firstly, it allows the application to dynamically set the item's MODIFIABLE attribute (thus allowing or preventing label editing from taking place) according to the particular context. Secondly, it gives the application a chance to save the original label in case it wishes to restore it later in the AFTER-EDIT event.

The AFTER-EDIT event has four options:

1. Do nothing, which case the new item label will be accepted.
2. Reject the new label, by restoring the previous value for the item's STRING attribute (as saved in the BEFORE-EDIT event).
3. Reject the new label, by setting the the item's STRING attribute to some other value that neither matches the new nor old label (e.g. silently „correcting“ the label entered by the user).
4. Re-enter edit mode for the item, forcing the user to modify the label again (possibly after displaying a message box to inform the user that the newly entered label is invalid).

As an example demonstrating some of the above topics, consider the following example. Firstly, we define some local data variables which we will need later:

```
01 #CONTROL HANDLE OF GUI 01 #ITEM HANDLE OF GUI
01 #LABEL (A) DYNAMIC 01 #POS (I4)
```

Having done this, we can can write a trivial BEFORE-EDIT event, where we simply save the existing label of the item about to be edited in the dynamic variable #LABEL:

```
#CONTROL := *CONTROL #ITEM := #CONTROL.ITEM #LABEL
:= #ITEM.STRING
```

To illustrate a few of the above techniques, we use the following AFTER-EDIT handler:

```
#CONTROL := *CONTROL #ITEM := #CONTROL.ITEM IF
#ITEM.STRING = ' ' #ITEM.STRING := #LABEL ELSE EXAMINE #ITEM.STRING TRANSLATE
INTO LOWER EXAMINE #ITEM.STRING FOR ' ' GIVING POSITION #POS IF #POS > 0 PROCESS
GUI ACTION CALL-DIALOG WITH #DLG$WINDOW #ITEM 'EDIT-LABEL' FALSE END-IF END-IF
```

The above code performs the following actions:

1. If the new item label consists only of blank, the old item label, as saved in the BEFORE-EDIT event is restored.
2. Otherwise, the new item label is converted into lower case (EXAMINE TRANSLATE).
3. Then, if the new item label contains a blank, we treat the data as invalid and raise an asynchronous user-defined event for the item in order to request corrected data from the user (more later).

The asynchronous event allows an invalid item label to be provisionally accepted. However, on receipt of the user-defined EDIT-LABEL event, we display a message box to inform the user that the data is invalid and in need of correction, then re-enter label editing mode. This is done via the following code in the DEFAULT event handler for the dialog:

```
IF *EVENT = 'EDIT-LABEL' #ITEM := *CONTROL OPEN
DIALOG NGU-MESSAGEBOX USING #ITEM.PARENT WITH #BUTTON 'Invalid data - please
re-enter'
'Label Edit' '!O' PROCESS GUI ACTION EDIT-LABEL WITH #ITEM GIVING *ERROR END-IF
```

If it is sufficient to set an edit mask and/or maximum label length in characters, and/or specify that only upper case characters should be allowed, this can be achieved without any coding by setting the EDIT-MASK attribute, LENGTH attribute or "Upper case (U)" STYLE flag (respectively) for the appropriate item(s). If an edit mask is specified, Natural automatically restores the old label, issuing a beep (if enabled), if the entered label does not match the mask.

Changing an Item's Label Programmatically

An item's label may be set directly via its STRING attribute. For example:

```
#ITEM.STRING := New label'
```

where #ITEM is the handle of the corresponding tree view or list view item.

In this case, only the item's edit mask (if any) is used. All other aspects of label editing described above do not apply here. In particular:

1. The label is changed regardless of the value of the MODIFIABLE attribute for the control and the item.
2. No BEFORE-EDIT or AFTER-EDIT events are raised.
3. The control's ITEM attribute is not set.
4. The text is not automatically translated to upper case if the item's "Upper case (U)" STYLE flag is set.
5. The supplied label can exceed the limit (if any) imposed by the item's LENGTH attribute.

91 Working with ActiveX Controls

- Terminology 690
- How To Define an ActiveX Control 690
- How To Create an ActiveX Control 690
- Accessing Simple Properties 691
- Colors 693
- Pictures 693
- Fonts 694
- Variants 695
- Arrays 696
- Using the PROCESS GUI Statement 697

ActiveX controls are third-party custom controls that you can integrate in a Natural dialog.

Terminology

ActiveX controls and Natural use different terminology in two cases:

ActiveX Control	Natural
Property	Attribute
Method	PROCESS GUI Statement Action

How To Define an ActiveX Control

The handle of an ActiveX control is defined similar as a built-in dialog element, but its individual aspects are coded in double angle brackets.

Example:

```
01    #OCX-1 HANDLE OF <<OCX-Table.TableCtrl.1 [Table Control]>>
```

In the above example, "Table.TableCtrl.1" is the program ID (ProgID) under which the ActiveX control is registered in the system registry. The prefix "OCX-" identifies the control as an ActiveX control. "[Table Control]" is an optional part of the definition and provides a readable name.

How To Create an ActiveX Control

You create an instance of an ActiveX control by using the `PROCESS GUI` statement action `ADD`. To do so, the value of the `TYPE` attribute must be the ActiveX control's ProgID prefixed with the string "OCX-" and put in double angle brackets. The ProgID is the name under which the control is registered in the system registry. You can additionally provide a readable name in square brackets. In addition to that, you can set Natural attributes such as `RECTANGLE-X` as well as the ActiveX control's properties. The property name must be preceded by the string "PROPERTY-" and this combination must be put in double angle brackets. Furthermore, you can suppress the ActiveX control's events. To do this, the event name must be preceded by the string "SUPPRESS-EVENT" this combination must be delimited by double angle brackets. The value of the `SUPPRESS-EVENT` property is either the Natural keyword "SUPPRESSED" or "NOT-SUPPRESSED".

Example:

```
PROCESS GUI ACTION ADD
    WITH PARAMETERS
       HANDLE-VARIABLE = #OCX-1
       TYPE = <<OCX-Table.TableCtrl.1 [Table Control]>>
       PARENT = #DLG$WINDOW
       RECTANGLE-X = 44
       RECTANGLE-Y = 31
       RECTANGLE-W = 103
       RECTANGLE-H = 46
       <<PROPERTY-HeaderColor>> = H'FF0080'
       <<PROPERTY-Rows>> = 16
       <<PROPERTY-Columns>> = 4
       <<SUPPRESS-EVENT-RowMoved>> = SUPPRESSED
       <<SUPPRESS-EVENT-ColMoved>> = SUPPRESSED
    END-PARAMETERS
```

Accessing Simple Properties

Simple properties are properties that do not have parameters. Simple properties of an ActiveX control are addressed like attributes of built-in controls. The attribute name is built by prefixing the property name with "PROPERTY-" and enclosing it in angle brackets.The properties of an ActiveX control are displayed in the Component Browser. The following examples assume that the ActiveX control `#OCX-1` has the simple properties `CurrentRow` and `CurrentCol`.

Example:

```
* Get the value of a property.
#MYROW := #OCX-1.<<PROPERTY-CurrentRow>>
* Put the value of a property.
#OCX-1.<<PROPERTY-CurrentCol>> := 17
```

The data types of ActiveX control properties are those defined by OLE Automation. In Natural, each of these data types is mapped to a corresponding Natural data type. The following table shows which OLE Automation data type is mapped to which Natural data type.

OLE Automation data type	NATURAL data type
VT_BOOL	L
VT_BSTR	A dynamic
VT_CY	P15.4
VT_DATE	T
VT_DECIMAL	Pn.m
VT_DISPATCH	HANDLE OF OBJECT
VT_ERROR	I4
VT_I1	I2
VT_I2	I2
VT_I4	I4
VT_INT	I4
VT_R4	F4
VT_R8	F8
VT_U1	B1
VT_U2	B2
VT_U4	B4
VT_UINT	B4
VT_UNKNOWN	HANDLE OF OBJECT
VT_VARIANT	(any Natural data type)
OLE_COLOR (VT_UI4)	B3
VT_FONT (VT_DISPATCH IFontDisp*)	HANDLE OF FONT HANDLE OF OBJECT (IFontDisp*) A dynamic
VT_PICTURE (VT_DISPATCH IPictureDisp*)	HANDLE OF OBJECT (IPictureDisp*) A dynamic

Read the table in the following way: Assume an ActiveX control #OCX-1 has a property named "Size", which is of type VT_R8. Then the expression #OCX-1.<<PROPERTY-SIZE>> has the type F8 in Natural.

Anmerkung: The Component Browser displays the corresponding Natural data types directly.

Some special data types are considered individually in the following:

Colors

A property of type Color appears in Natural as a B3 value. The B3 value is interpreted as an RGB color value. The three bytes contain the red, green and blue elements of the color, respectively. Thus for example H'FF0000' corresponds to red, H'00FF00' corresponds to green, H'0000FF' corresponds to blue and so on.

Example:

```
...
 01 #COLOR-RED (B3)
...
 #COLOR-RED := H'FF0000'
 #OCX-1.<<PROPERTY-BackColor>> := #COLOR-RED
...
```

Pictures

A property of type Picture appears in Natural as `HANDLE OF OBJECT`. Alternatively you can assign an Alpha value to a Picture property. The Alpha value must then contain the file name of a Bitmap (*.bmp*) file.

Example (usage of Picture properties):

```
...
 01 #MYPICTURE HANDLE OF OBJECT
...
 * Assign a Bitmap file name to a Picture property.
 #OCX-1.<<PROPERTY-Picture>>:= '11100102.bmp'
 *
 * Get it back as an object handle.
 #MYPICTURE := #OCX-1.<<PROPERTY-Picture>>
 *
 * Assign the object handle to a Picture property of another control.
 #OCX-2.<<PROPERTY-Picture>>:= #MYPICTURE
...
```

Fonts

A property of type Font appears in Natural as `HANDLE OF OBJECT`. You can alternatively assign a `HANDLE OF FONT` to a Font property. Additionally you can assign an Alpha value to a Font property. The Alpha value must then contain a font specification in the form that is returned by the `STRING` attribute of a `HANDLE OF FONT`.

Example 1 (using `HANDLE OF OBJECT`):

```
...
01 #MYFONT HANDLE OF OBJECT
...
* Create a Font object.
CREATE OBJECT #MYFONT OF CLASS 'StdFont'
#MYFONT.Name := 'Wingdings'
#MYFONT.Size := 20
#MYFONT.Bold := TRUE
*
* Assign the Font object as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #MYFONT
...
```

Example 2 (using `HANDLE OF FONT`):

```
...
01 #FONT-TAHOMA-BOLD-2 HANDLE OF FONT
...
* Create a Font handle.
PROCESS GUI ACTION ADD WITH PARAMETERS
    HANDLE-VARIABLE = #FONT-TAHOMA-BOLD-2
    TYPE = FONT
    PARENT = #DLG$WINDOW
    STRING = '/Tahoma/Bold/0 x -27/ANSI VARIABLE SWISS DRAFT/W/2/3/'
END-PARAMETERS GIVING *ERROR
...
* Assign the Font handle as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #FONT-TAHOMA-BOLD-2
...
```

Example 3 (using a font specification string):

```
...
01 #FONT-TAHOMA-BOLD-2 HANDLE OF FONT
...
* Create a Font handle.
PROCESS GUI ACTION ADD WITH  PARAMETERS
    HANDLE-VARIABLE = #FONT-TAHOMA-BOLD-2
    TYPE = FONT
    PARENT = #DLG$WINDOW
    STRING = '/Tahoma/Bold/0 x -27/ANSI VARIABLE SWISS DRAFT/W/2/3/'
END-PARAMETERS GIVING *ERROR
...
* Assign the font specification as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #FONT-TAHOMA-BOLD-2.STRING
...
```

Variants

A property of type Variant is compatible with any Natural data type. This means that the type of the expression `#OCX-1.<<PROPERTY-Value>>` is not checked by the compiler, if "Value" is a property of type Variant. So the assignments `#OCX-1.<<PROPERTY-Value >> := #MYVAL` and `#MYVAL := #OCX-1.<<PROPERTY-Value >>` are allowed independently of the type of the variable `#MYVAL`. It is however up to the ActiveX control to accept or reject a particular property value at runtime, or to deliver the value in the requested format. If it does not, the ActiveX control will usually raise an exception. This exception is returned as a Natural error code to the Natural program. Here it can be handled in the usual way in an `ON ERROR` block. You should check the documentation of the ActiveX control to find out which data formats are actually allowed for a particular property of type Variant.

An expression like `#OCX-1.<<PROPERTY-Value>>` (where "Value" is a Variant property) can occur as source operand in any statement. However, it can be used as target operand only in assignment statements.

Examples (usage of Variant properties):

(Assume that "Value" is a property of type Variant of the ActiveX control #OCX-1)

```
...
01 #STR1 (A100)
01 #STR2 (A100)
...
* These statements are allowed, because the Variant property is used
* as source operand (its value is read).
#STR1 := #OCX-1.<<PROPERTY-Value>>
COMPRESS #OCX-1.<<PROPERTY-Value>> 'XYZ' to #STR2
...
* This leads to an error at compiletime, because the Variant
* property is used as target operand (its value is modified) in
* a statement other than an assignment.
COMPRESS #STR1 "XYZ" to #OCX-1.<<PROPERTY-Value>>
...
* This statement is allowed, because the Variant property is used
* as target operand in an assignment.
COMPRESS #STR1 'XYZ' to #STR2
#OCX-1.<<PROPERTY-Value>> := #STR2
...
```

Arrays

A property of type SAFEARRAY of up to three dimensions appears in a Natural program as an array with the same dimension count, occurrence count per dimension and the corresponding format. (Properties of type SAFEARRAY with more than three dimensions cannot be used in Natural programs.) The dimension and occurrence count of an array property is not determined at compiletime but only at runtime. This is because this information is variable and is not defined at compiletime. The format however is checked at compiletime.

Array properties are always accessed as a whole. So no index notation is necessary and allowed with an array property.

Examples (usage of Array properties):

(Assume that "Values" is a property of the ActiveX control `#OCX-1` an has the type SAFEARRAY of VT_I4)

```
...
01 #VAL-L (L/1:10)
01 #VAL-I (I4/1:10)
...
* This statement is allowed, because the format of the property
* is data transfer compatible with the format of the receiving array.
* However, if it turns out at runtime that the dimension count or
* occurrence count per dimension do not match, a runtime error will
* occur.
VAL-I(*) := #OCX-1.<<PROPERTY-Values>>
...
* This statement leads to an error at compiletime, because
* the format of the property is not data transfer compatible with
* the format of the receiving array.
VAL-L(*) := #OCX-1.<<PROPERTY-Values>>
...
```

Using the PROCESS GUI Statement

The methods of ActiveX controls are called as actions in a `PROCESS GUI` statement. The same is the case with the complex properties of ActiveX controls (i. e. properties that have parameters). The methods and properties of an ActiveX control are displayed in the Component Browser.

This section covers the following topics:

- Performing Methods
- Getting Property Values
- Putting Property Values
- Optional Parameters
- Error Handling
- Using Events With Parameters

- Suppressing Events At Runtime

Performing Methods

To perform a method of an ActiveX control the PROCESS GUI statement is used. The name of the corresponding PROCESS GUI action is built by prefixing the method name with "METHOD-" and enclosing it in angle brackets. The ActiveX control handle and the method parameters (if any) are passed in the WITH clause of the PROCESS GUI statement. The return value of the method (if any) is received in the variable specified in the USING clause of the PROCESS GUI statement.

This means: To perform a method, you enter a statement

PROCESS GUI ACTION <<METHOD-*methodname*>> WITH*handlename* [*parameter*]...
[USING *method-return-operand*]..

Examples:

```
* Performing a method without parameters:
PROCESS GUI ACTION <<METHOD-AboutBox>> WITH #OCX-1
* Performing a method with parameters:
PROCESS GUI ACTION <<METHOD-CreateItem>> WITH #OCX-1 #ROW #COL #TEXT
* Performing a method with parameters and a return value:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN
```

Formats and length of the method parameters and the return value are checked at compiletime against the definition of the method, as it is displayed in the Component Browser.

Getting Property Values

To get the value of a property that has parameters, the name of the corresponding PROCESS GUI action is built by prefixing the property name with "GET-PROPERTY-" and enclosing it in angle brackets. The ActiveX control handle and the property parameters (if any) are passed in the WITH clause of the PROCESS GUI statement. The property value is received in the USING clause of the PROCESS GUI statement.

This means: To get the value of a property that has parameters, you enter a statement

```
PROCESS GUI ACTION <<GET-PROPERTY-propertyname>> WITHhandlename[parameter]
... USING get-property-operand
```

Example:

```
PROCESS GUI ACTION <<GET-PROPERTY-ItemHeight>> WITH #OCX-1 #ROW #COL USING #ITEMHEIGHT
```

Formats and length of the property parameters and the property value are checked at compiletime against the definition of the method, as it is displayed in the Component Browser.

Putting Property Values

To put the value of a property that has parameters, the name of the corresponding `PROCESS GUI` action is built by prefixing the property name with "PUT-PROPERTY-" and enclosing it in angle brackets. The ActiveX control handle and the property parameters (if any) are passed in the `WITH` clause of the `PROCESS GUI` statement. The property value is passed in the `USING` clause of the `PROCESS GUI` statement.

This means: To put the value of a property that has parameters, you enter a statement

```
PROCESS GUI ACTION <<PUT-PROPERTY-propertyname>> WITHhandlename[parameter]
... USING put-property-operand
```

Example:

```
PROCESS GUI ACTION <<PUT-PROPERTY-ItemHeight>> WITH #OCX-1 #ROW #COL USING #ITEMHEIGHT
```

Formats and length of the property parameters and the property value are checked at compiletime against the definition of the method, as it is displayed in the Component Browser.

Optional Parameters

Methods of ActiveX controls can have optional parameters. This is also true for parameterized properties. Optional parameters need not to be specified when the method is called. To omit an optional parameter, use the placeholder 1X in the `PROCESS GUI` statement. To omit *n* optional parameters, use the placeholder *n*X.

In the following example it is assumed that the method `SetAddress` of the ActiveX control `#OCX-1` has the parameters `FirstName`, `MiddleInitial`, `LastName`, `Street` and `City`, where `MiddleInitial`, `Street` and `City` are optional. Then the following statements are correct:

```
* Specifying all parameters.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName Street City
* Omitting one optional parameter.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName 1X LastName Street City
* Omitting the optional parameters at end explicitly.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName 2X
* Omitting the optional parameters at end implicitly.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName
```

Omitting a non-optional (mandatory) parameter results in a syntax error.

Error Handling

The `GIVING` clause of the `PROCESS GUI` statement can be used as usual to handle error conditions. The error code can either be caught in a user variable and then be handled, or the normal Natural error handling can be triggered and the error condition be handled in an `ON ERROR` block.

Example:

```
DEFINE DATA LOCAL
1 #RESULT-CODE (N7)
...
END-DEFINE
...
* Catching the error code in a user variable:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN GIVING
#RESULT-CODE
*
* Triggering the Natural error handling:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN GIVING
*ERROR-NR
...
```

Special error conditions that can occur during the execution of ActiveX control methods are:

- A method parameter, method return value or property value could not be converted to the data format expected by the ActiveX control. (These format checks are normally already done at compiletime. In these cases no runtime error can be expected. However, note that method parameters, method return values or property values defined as Variant are not checked at

compiletime. This applies also for arrays and for those data types that can be mapped to several possible Natural data types.)

- A COM or Automation error occurs while locating and executing a method.
- The ActiveX control raises an exception during the execution of a method.

In these cases the error message contains further information provided by the ActiveX control, which can be used to determine the reason of the error with the help of the documentation of the ActiveX control.

Using Events With Parameters

Events sent by ActiveX controls can have parameters. In the controls event-handler sections, these parameters can be queried. Parameters passed by reference can also be modified. The events of an ActiveX control, the names and data types of the parameters and the fact if a parameter is passed by value or by reference is all displayed in the Component Browser.

Event parameters of an ActiveX control are addressed like attributes of built-in controls. The attribute name is built by prefixing the parameter name with "PARAMETER-" and enclosing it in angle brackets. Alternatively, parameters can be addressed by position. This means the attribute name is built by prefixing the number of the parameter with "PARAMETER-" and enclosing it in angle brackets. The first parameter of an event has the number 1, the second the number 2 and so on. These attribute names are only valid inside the event handler of that particular event.

In the following examples it is assumed that a particular event of the ActiveX control `#OCX-1` has the parameters `KeyCode` and `Cancel`. Then the event handler of that event might contain the following statements:

```
* Querying a parameter by name:
#PRESSEDKEY := #OCX-1.<<PARAMETER-KeyCode>>
* Querying a parameter by position:
#PRESSEDKEY := #OCX-1.<<PARAMETER-1>>
```

Parameters that are passed by reference can be modified in the event handler. In the following example it is assumed that the `Cancel` parameter is passed by reference and is thus modifiable. Then the event handler might contain the following statements:

```
* Modifying a parameter by name:
#OCX-1.<<PARAMETER-Cancel>>:= TRUE
* Modifying a parameter by position:
#OCX-1.<<PARAMETER-2>>:= TRUE
```

Suppressing Events At Runtime

To suppress or unsuppress an event of an ActiveX control at runtime, modify the corresponding suppress event attribute of the control. The name of the suppress event attribute is built by prefixing the event name with "SUPPRESS-EVENT-" and enclosing it in angle brackets. The events of an ActiveX control are displayed in the Component Browser.

The following example assumes that the ActiveX control `#OCX-1` has the event `ColMoved`.

```
* Suppress the event.
#OCX-1.<<SUPPRESS-EVENT-ColMoved>> := SUPPRESSED
* Unsuppress the event.
#OCX-1.<<SUPPRESS-EVENT-ColMoved>> := NOT-SUPPRESSED
```

92 Working with Arrays of Dialog Elements

It is sometimes convenient to arrange dialog elements in one or two dimensions. If, for example, you want to arrange several radio button controls in one column, it is possible to draw the first one and specify the others as a one-dimensional array.

To work with arrays of dialog elements

1. Choose the **Array** button in the radio button control's attributes window. The **Array Specification** dialog box appears.
2. Enter:
   - the number of dimensions;
   - the bounds of the first and second dimension, if applicable;
   - the spacing on the x and y axis in pixels (depending on whether the array is arranged in rows or in columns);
   - the arrangement (rows or columns).

The array will now be treated as a graphical entity. Note that you will have to assign a common `GROUP-ID` attribute to each radio button control. This will enable you to treat the array as a logical entity.

For each dialog element in an array, the following attributes may be specified separately:

- `STRING`
- `DIL-TEXT`
- `BITMAP-FILE-NAME`

In an event handler for an array of dialog elements, the system variable `*CONTROL` will denote one of the array elements.

If a variable is selected as the source of an attribute value, the array must contain at least the index ranges of the dialog element.

If a message file ID is specified as the source of an attribute value, consecutive messages are taken for the array's sequence of dialog elements.

In an array of dialog elements, you can assign one value to all dialog elements in the array using the (*) notation or a range, such as in the following examples:

```
#PB-1.ENABLED(*) := TRUE    /*invalid
#PB-1.ENABLED(1:3) := TRUE  /*invalid
```

An alternative way of creating a sequence of identical dialog elements is to duplicate or copy and paste an individual dialog element and use the grid plus the cross-hair cursor to place them.

The following example illustrates how to set the STRING attribute of occurence 2 in a one-dimensional push button array:

```
#PB-2.STRING(2) := 'HUGO'
```

93 Working with Control Boxes

- Introduction 706
- Purpose of Exclusive Control Boxes 706
- Examples of Use of Exclusive Control Boxes 707
- Creation of the Wizard Pages 708

Introduction

A control box is is used to enhance the effectiveness of the nested control support. However, control boxes have a number of unique features that merit their separate discussion.

Control boxes are, in themselves, fairly inert controls, belonging to the same category as text constants and group frames in that they cannot receive the focus and do not receive any mouse or keyboard input. Instead, they are intended to act as general-purpose containers for other controls (including, possibly, other control boxes), in order to build up a control hierarchy. In doing so, control boxes support three styles which are worthy of special mention here:

- Because it is often desirable to be able to group controls together for convenience, but not desirable that the user actually sees the container itself, control boxes can be marked with the style "transparent". In this case, no parts of the control box are drawn, and any underlying colors and controls show through.
- Control boxes can also be marked with the style "exclusive". When an exclusive control box is made visible, either in the dialog editor or at runtime, all other sibling control boxes that are also marked as "exclusive" are hidden. This applies to edit-time and runtime in a slightly different way. At runtime, setting the `VISIBLE` attribute of an exclusive control box to `TRUE` hides all its exclusive siblings and sets their `VISIBLE` attribute to `FALSE`. At edit-time, whenever an exclusive control box or one of its descendants is selected, the exclusive control box becomes visible and all other exclusive siblings are hidden. However, in this latter case the `VISIBLE` attribute of the controls concerned is unaffected. This implies that the exclusive control box that is initially visible when the dialog is run is independent of the exclusive control box that was visible at the time the dialog was last saved.
- Additionally, control boxes support the "size to parent" style. When a container control, or the dialog itself, is resized, all child control boxes (if any) with this style set are resized to entirely fill the parent's client area. The same applies when this style is first set in the dialog editor. However, it is still possible to resize such control boxes independently of their container.

Purpose of Exclusive Control Boxes

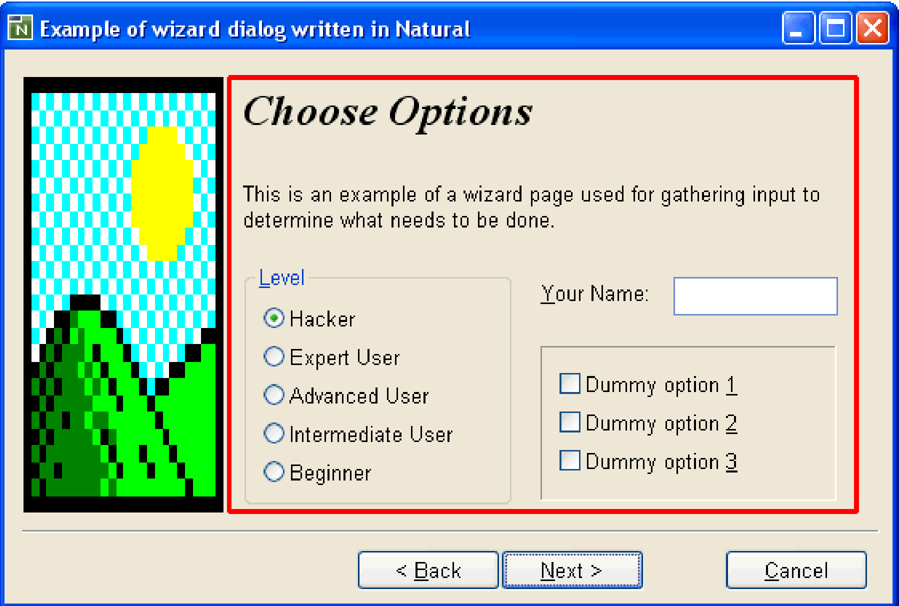
Exclusive control boxes, as described above, are primarily intended for situations where it is necessary to manage several overlapping „pages“ of controls occupying the same region of a dialog. Without the auto-hiding feature which exclusive control boxes provide, it would be very difficult indeed for a user to handle this situation in the dialog editor, as many controls would be partially or completely overlapped by others. Of course, one could move the control to the front of the control sequence during editing, but this would be highly inconvenient, and one would have to remember to move the control back before continuing.

Using exclusive control boxes, editing a control in this situation is as simple as selecting it. For controls that are not currently on display, the selection can be made via the combo box in the dialog editor's status bar or by using the TAB key to walk through the controls sequentially until the target control is reached. When a control that is a descendant of an exclusive control box is selected, that exclusive control box is made visible (if not already so), and the previously visible exclusive control box is hidden. These changes have no impact on the generated dialog source code and the runtime state of the dialog.

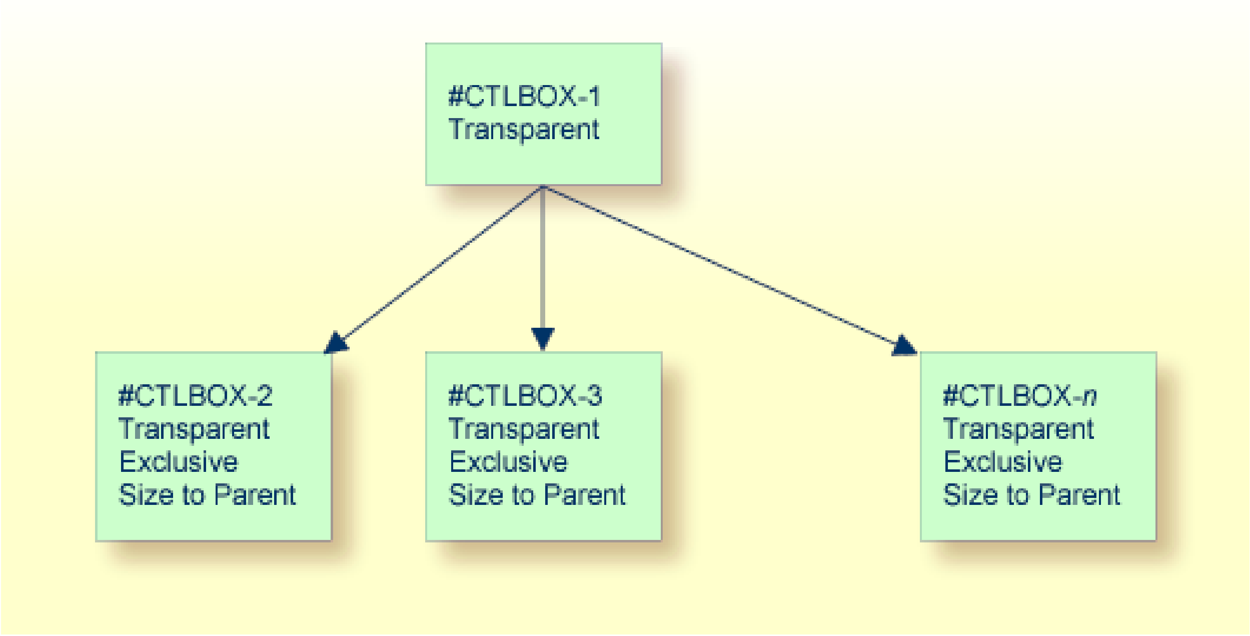
Examples of Use of Exclusive Control Boxes

Although the design of control boxes was intended to keep them as general as possible, two possible situations where overlapping control pages are desired (and hence where exclusive control boxes become extremely useful) are worthy of special mention here:

- Wizard dialogs.
- Tabbed dialogs („Property sheets“).



Within the rectangle highlighted in red, the so-called "wizard pages" are displayed. Within this area, we use a 2-level hierarchy of control boxes in order to implement the required functionality:



Here, `#CTLBOX-1` is used as the „master“ control box, which makes resizing of the pages easier later, should this become necessary. Because all child control boxes are marked with the style "size to parent", we can resize the wizard page area simply by resizing `#CTLBOX-1`.

The child control boxes are used to implement the actual wizard pages. `#CTLBOX-2` contains the controls used for wizard page 1, `#CTLBOX-3` contains the controls for wizard page 2, and so on.

Creation of the Wizard Pages

Creation of the wizard pages typically involves the following steps:

1. Create the top-level („master“) control box as for any other control.
2. Via its attributes window, set the "transparent" style.
3. Create another control box within the first one. The new control box automatically becomes a child of the first one, because control boxes are always containers.
4. Via the attributes window for the child control box, set the "transparent", "exclusive" and "size to parent" styles. Because the "size to parent" style is set, the child control box expands to fill its container.
5. Now you can start adding the controls onto the newly-created control box, which becomes wizard page 1.

6. Adding a new wizard page is most easily achieved by selecting the child control box you wish to immediately precede the new one, then using the clipboard copy and paste commands. Before doing the copy, Natural will prompt you as to whether you want the child controls to be copied, too. Answer this question with **No**.
7. Because the newly added child control box also has the exclusive flag set, the previously displayed child control box is hidden, and the new blank one is shown, ready for you to start adding a new set of controls as for the first wizard page.

Switching between the wizard pages at edit-time

Switching between the pages at edit time can be most simply achieved by selecting the child control box for the appropriate page, or one of the controls on it, from the combo box in the dialog editor's status bar.

Creating the divider line

The divider line between the push buttons and the wizard pages can be implemented as a very thin group box (2 pixels high) with no caption. The still slightly visible sides of the group box at each end can be masked out by using a transparent control box which comes after the group frame in the control sequence. Make sure the "control clipping" style for the dialog is switched on for this technique to work.

Implementing the Back and Next push buttons

Firstly, define a local variable for the dialog to store the handle of the currently active page. E.g.:

```
01 #ACTPAGE HANDLE OF CONTROLBOX ...
```

Secondly, set this variable to the handle of the first wizard page in the `AFTER-OPEN` event for the dialog:

```
#ACTPAGE := #CTLBOX-1.FIRST-CHILD ..
```

where #CTLBOX-1 is the handle of the top-level control box. Now we are ready to implement the CLICK event code for the **Next** push button (#PB-NEXT). This could look something like this:

```
IF #ACTPAGE.SUCCESSOR = NULL-HANDLE
  CLOSE DIALOG *DIALOG-ID
ELSE
  REPEAT
    #ACTPAGE := #ACTPAGE.SUCCESSOR
    WHILE #ACTPAGE.ENABLED = FALSE
  END-REPEAT
  #ACTPAGE.VISIBLE := TRUE
  IF #ACTPAGE.SUCCESSOR = NULL-HANDLE
    #PB-NEXT.STRING := 'Finish'
    #PB-BACK.ENABLED := FALSE
    #PB-CANCEL.ENABLED := FALSE
  ELSE
    #PB-BACK.ENABLED := TRUE
  END-IF
END-IF
..
```

Note that this logic does not be modified if further wizard pages are added later. Note also that any intermediate wizard pages whose corresponding control box has been disabled are ignored. This allows certain wizard pages to be skipped, based on previous input, by simply setting the relevant control box ENABLED attribute to FALSE. When the last page is reached, the text for the **Next** push button is changed to "Finish".

The CLICK event code for the **Back** push button (#PB-BACK) is very similar:

```
REPEAT
  #ACTPAGE := #ACTPAGE.PREDECESSOR
  WHILE #ACTPAGE.ENABLED = FALSE
END-REPEAT
IF #ACTPAGE.PREDECESSOR = NULL-HANDLE
  #PB-BACK.ENABLED := FALSE
END-IF
#ACTPAGE.VISIBLE := TRUE
..
```

Note that the **Back** push button should be initially disabled in the dialog editor.

Clearing all controls on a wizard page

This can be conveniently achieved by selecting any (highest-level) control on the relevant page, then performing a **Select All** from the **Edit** menu to additionally select all the controls siblings. The selected controls can then be deleted as normal.

Example 2 - a tabbed dialog

A tabbed dialog (sometimes called a „property sheet“) is very similar in concept to a wizard dialog. The only substantial difference is that instead of navigating between the control „pages“ via the **Next** and **Back** push buttons, the user directly accesses the page he wants by clicking on the appropriate tab. The control page hierarchy can be built up and handled in the dialog editor in the same way as in the wizard dialog example above. Several ActiveX controls are available which provide the actual tabs.

It should be noted, however, that the switching between the pages (i.e., switching between the corresponding control boxes) is not automatic. The Natural programmer must insert code for the ActiveX event raised by a tab switch, find out which tab is selected, and set the `VISIBLE` attribute of the appropriate (exclusive) control box to `TRUE`. This cannot be done implicitly by Natural because each ActiveX control can implement its functionality in any way it chooses. There is no standard event raised for a tab switch and no standard method with standard parameters (or standard property) for determining the currently active tab.

An example tabbed dialog, making use of the Microsoft "Tab Strip" ActiveX control (*V4-NEST.NS3*) is shipped as part of the Natural example libraries.

94 Working with Date and Time Picker (DTP) Controls

- Introduction 714
- Date and Time Formats 714
- Inputting Dates and Times 715
- Null Values 716
- Calendar Colors and Font 716

Introduction

A date and time picker (DTP) control is used to simplify the input of date or time information for the user. A DTP control appears and behaves similarly to a spin control for the input of times and optionally as either a spin control or selection box for the input of dates. In the latter case, a month calendar appears instead of the typical list box when the user clicks on the button displaying the down arrow.

Date and Time Formats

By default, the date and time information is displayed according to the date and time formats defined for the current regional settings. Because Windows provides two alternative date formats, one long and one short (both of which may be changed by the user), and because the short date format may not contain century information, one of three `STYLE` flags determines which of the standard date formats should be used. These (mutually exclusive) formats are:

- "Short date (s)", implying that the standard short date format for the current regional settings should be used.
- "Century date (c)", implying that the standard short date format for the current regional settings should be used, but extended to provide century information if this is not already the case. Note that in many cases, the short date format already includes century information, in which case this style does not change the appearance of the date.
- "Long date (d)", implying that the standard long date format for the current regional settings should be used

In addition. The "Time (t)" style flag is provided in order to indicate that the control should display time (instead of date) information.

If these standard formats are not sufficient, they can be overridden by proving a custom format string using the `EDIT-MASK` attribute. Note, however, that the format string specifiers do not correspond to those used for edit masks elsewhere within Natural. The following table lists the available specifiers and their meanings:

Specifier	Description
d	The one- or two-digit day.
dd	The two-digit day. Single-digit day values are preceded by a zero.
ddd	The three-character weekday abbreviation.
dddd	The full weekday name.
h	The one- or two-digit hour in 12-hour format.

Specifier	Description
hh	The two-digit hour in 12-hour format. Single-digit values are preceded by a zero.
H	The one- or two-digit hour in 24-hour format.
HH	The two-digit hour in 24-hour format. Single-digit values are preceded by a zero.
m	The one- or two-digit minute.
mm	The two-digit minute. Single-digit values are preceded by a zero.
s	The one- or two-digit second.
ss	The two-digit second. Single-digit values are preceded by a zero.
M	The one- or two-digit month number.
MM	The two-digit month number. Single-digit values are preceded by a zero.
MMM	The three-character month abbreviation.
MMMM	The full month name.
t	The one-letter AM/PM abbreviation (that is, AM is displayed as "A").
tt	The two-letter AM/PM abbreviation (that is, AM is displayed as "AM").
yy	The last two digits of the year (that is, 2005 would be displayed as "05").
yyyy	The full year (that is, 2005 would be displayed as "2005").

In addition, any characters in quotes are displayed exactly as specified. To specify the quote character itself within a quoted string, two consecutive single quote characters should be used. Spaces and punctuation marks (such as the comma) do not need to be quoted.

For example, in order to display the string "John's birthday is Friday, December 31, 1969", the DTP control's `EDIT-MASK` attribute would be set to "John' 's birthday is' dddd, MMMM d, yyyy".

Inputting Dates and Times

The DTP control provides several ways of modifying the specified information:

- By the user, by entering numerical information (day numbers, etc.) directly.
- By the user, by incrementing or decrementing the selected field (e.g. day number, month name) via the + or - keys, respectively.
- By the user, if the DTP control has either the "Time (t)" or "Up-down (u)" style, by selecting the required field and incrementing or decrementing the value via the up-down ("spin") control.
- By the user, if the DTP control is using a month calendar, by pressing the down arrow to open the month calendar and navigating to the required date. Unlike the above method, this method updates all date fields simultaneously.
- Programmatically, by updating the `TIME` attribute with the required date or time.

For example, to set the date or time in a DTP control to the current date or time, use the following assignments:

```
#DTP-1.TIME := *DATX
```

or

```
#DTP-1.TIME := *TIMX
```

respectively, where `#DTP-1` is assumed to be the handle of the DTP control.

Note that the DTP control stores both date and time information, even though it only allows editing of the date or time component, depending on the control's style.

If the DTP control's date or time is modified by the user, a `CHANGE` event is raised for the control (if not suppressed). This does not happen if the DTP control is modified programmatically.

Null Values

If the "Allow 'no value' (n)" style is specified for the DTP control, the control displays a check box. If this check box is unchecked, the interpretation is that there is no date or time associated with the control. The application can test for this state by querying the control's `CHECKED` attribute. It can also revert the control to the "no value" state by setting the `CHECKED` attribute back to `UNCHECKED`. Note that it is, however, not possible to explicitly set the `CHECKED` attribute to `CHECKED`, as this is done implicitly whenever a date or time is applied to the control. Furthermore, the `CHECKED` attribute may not be set at all for DTP controls without the "Allow 'no value' (n)" style.

Calendar Colors and Font

The colors and font used by the month calendar (if any) associated with the DTP control may be changed by use of the `SET-AUX-COLOR` and `SET-AUX-FONT` actions, respectively.

95 Working with Dialog Bar Controls

- Introduction 718
- Creating a Dialog Bar Control 718
- Types of Dialog Bar Control 718
- UI Transparency 722
- Client-Size Event 722
- Close Button 722
- Sample Code 722

Introduction

A dialog bar is similar to a tool bar control in that it can either docked to one of the interior sides of the dialog's frame or (optionally) floated in its own separate window. Unlike tool bar controls, however, dialog bar controls are conceived general-purpose container controls and are not dedicated to containing primarily tool bar items. Furthermore, there are a number of other visual and behavioral differences between tool bar controls and dialog bars, some of which are discussed below.

A good example of a dialog bar control is the library workspace window in Natural Studio.

Creating a Dialog Bar Control

Dialog bar controls are created in the dialog editor in the same way as other standard controls (such as list boxes or push buttons) are. That is, they are either created statically in the dialog editor via the **Insert** menu or by drag and drop from the Insert tool bar, or dynamically at runtime by using a `PROCESS GUI ACTION ADD` statement with the `TYPE` attribute set to `DIALOGBAR`.

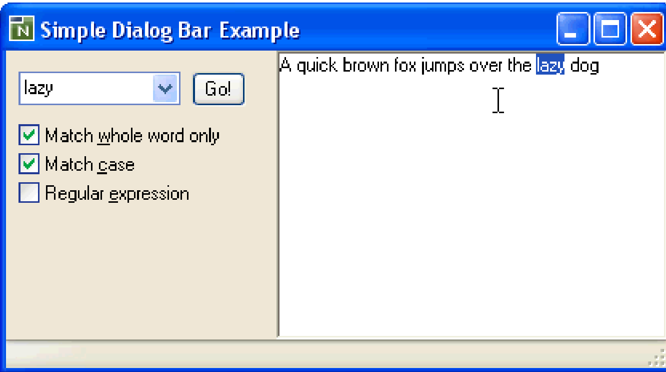
Types of Dialog Bar Control

A dialog bar control can exist in one of the following three basic forms (in order of complexity):

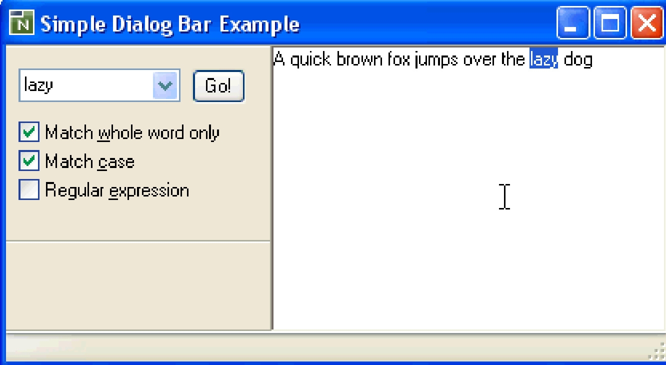
1. Neither dockable nor sizeable.
2. Dockable, but not sizeable.
3. Dockable and sizeable.

The dialog bar control is dockable if its `DRAGGABLE` attribute is set. It is sizeable if the "Dynamic (Y)" `STYLE` flag is set.

The following example shows an example of a non-dockable, non-sizeable dialog bar. The edit area on the right fills the entire client area of the dialog. The dialog bar cannot be dragged by the user and extends to fill the entire length of the side on which it is positioned:

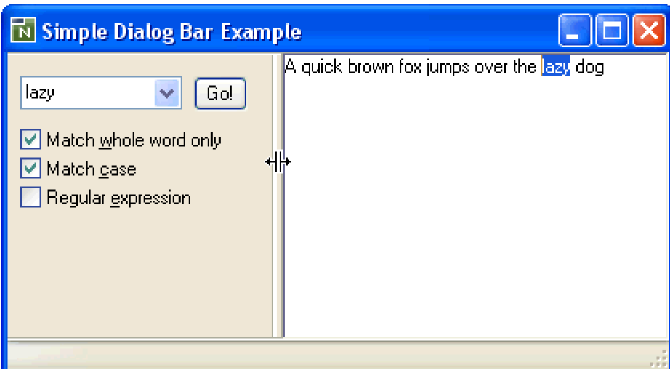


Setting the „dockable“ state in the **Dialog Bar Attributes** window in the dialog editor causes the window to be draggable by the user. Note also that the dialog bar no longer extends to occupy the full length of the side to which it is docked (another dialog bar control or tool bar control could be docked underneath it):

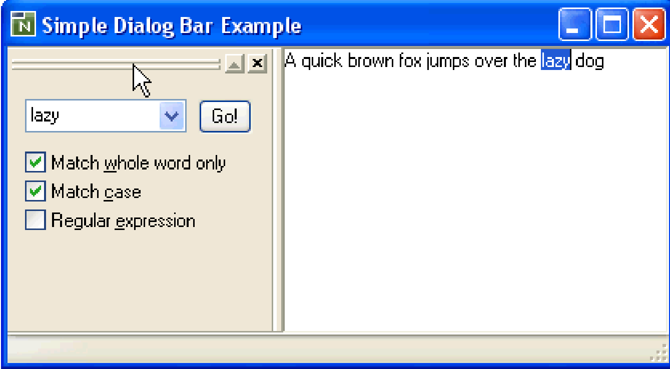


The user can drag the dialog bar control and re-dock it to another side of the owner dialog, or float it in its own separate window, as shown below:

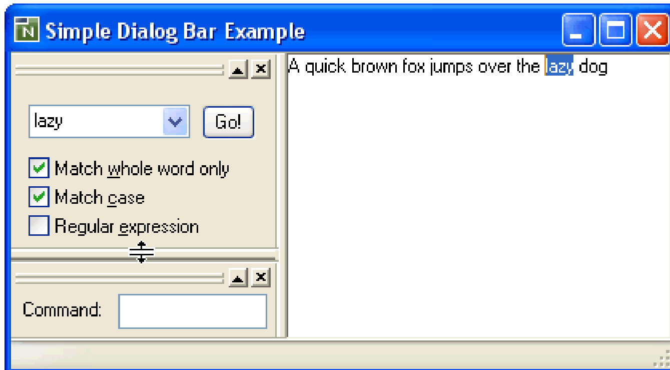
If the dialog bar control is made sizeable (by checking the "Dynamic (Y)" style flag in the **Attributes** window), a longitudinal splitter bar appears, allowing the dialog bar control to be resized. Note that sizeable dialog bar controls expand to fill the entire length of the side they are docked to that is not occupied by non-sizeable bars:



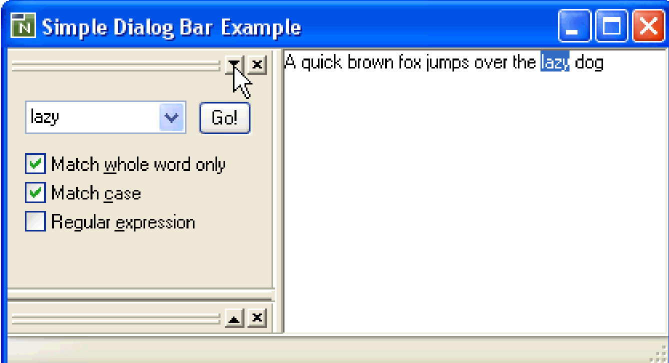
If a gripper bar, zoom and close button are added (by setting the "Gripper (g)", "Zoom button (z)" and "Close button (x)" style flags in the **Attributes** window), the dialog bar control takes on the familiar appearance of the control used to display the library workspace in Natural Studio. Note that the zoom button is disabled, because there is no other sizeable dialog bar control on the same row:



If a second sizeable dialog bar control is added, and docked alongside the first on the same row, a transverse splitter bar appears allowing the relative sizes of the two dialog bar controls to be changed. Note that the zoom button is now enabled:



Clicking on the zoom button toggles between the maximized and restored states of a sizeable dialog bar control. Maximizing a dialog bar control causes the other sizeable dialog bar controls on the same row to be minimized, and the released space to be taken up by the maximized bar, as shown below:



Note that the direction of the arrow is displayed by the zoom button on the maximized bar has changed direction in order to indicate that the next time this button is pressed, the bar will be restored, rather than maximized. When a bar is restored, all sizeable dialog bars on the row revert to their normalized sizes. These are usually the sizes prior to the maximize operation, unless there has been a change in the visible bars on the row in the meantime (e.g., visible bar hidden or hidden bar shown, new bar docked on row, etc.).

Note that if the length of a bar on a row is changed via a tranverse splitter bar, all visible bars on the row are automatically restored.

UI Transparency

A dockable dialog bar control may normally be dragged via either its gripper bar (if any) or its background. If the "UI transparent (T)" style is set, however, the bar may only be dragged via its gripper bar (if any). If such a (sizeable) bar does not have a gripper bar, resizing of the control is only possible via the splitter bar(s), which may be a desirable feature in some situations. Additionally, only allowing dragging via the gripper bar helps prevent unintentional initiation of drag operations.

Client-Size Event

The dialog's client window is reduced in size to exclude the areas occupied by tool bar, status bar and dialog bar controls. If a dockable window (tool bar or dialog bar control) is floated, re-docked to another side of the owner window, or is shown or hidden, the size of the client window can change, even though the exterior dimensions of the window have not altered. Because the `SIZE` event is reserved for changes in a dialog's exterior size, applications which need to keep track of the size of the client window should instead use the `CLIENT-SIZE` event for this purpose. The actual size of the dialog client window can then be determined within this event by means of the `INQ-INNER-RECT` action.

Close Button

The close button (if present) hides a dialog bar control rather than closing it, as is also the case for the close button on floated tool bar controls. It is up to the application to provide a method of re-showing the bar. The next section provides some code for doing this (amongst other things).

Sample Code

Below is a full listing of an external subroutine that can, in most cases, be used „as-is“ in order to allow user control over the display of tool bars and dialog bars. The code is designed to be powerful enough to cope with MDI applications, but also works with non-MDI (i.e., SDI) applications.

The subroutine appends the tool and dialog bar captions (`STRING` attribute) to the dialog's context menu. If the dialog does not have a context menu, one is created and assigned to the dialog automatically. It assumes that, in an MDI application, there are some tool bars and dialog bars that are global (i.e., relevant for all types of MDI child dialogs) and some which are private (i.e., relevant only for one type of MDI child dialog). For example, in Natural Studio, the Object tool bar is an

example of a global tool bar, whereas the dialog editor options tool bar is private to the dialog editor. When the user switches between MDI child dialogs, the context menu is changed to only show the global tool bars plus any private tool bars relevant to the currently active dialog. Furthermore, the same private bars are displayed as the last time this dialog was displayed (if the **Save layout** check box in the **Dialog Attributes** window is checked, the subset of bars shown is even retained between sessions).

The subroutine should be called in the `AFTER-ANY` event handler of the main dialog (i.e., the MDI frame dialog for MDI applications), as follows (assuming the main dialog's handle variable name is set to the default value of `#DLG$WINDOW`):

```
PERFORM PROCESS-BAR-COMMANDS #DLG$WINDOW
```

In addition, the following steps are optional:

1. The bars are listed in the context menu in the order in which they appear in the control sequence. Therefore, you may wish to re-sequence the tool and dialog bars (e.g., to ensure that the global tool bars are displayed before the private ones in MDI applications).
2. The code does not insert a separator before the list of available bars on the context menu. Therefore, if you are already using a context menu for the dialog, you would probably want to ensure that your context menu ends with a separator.
3. For MDI applications, for each private bar, you should enter the name of the dialog (e.g. "CHILD" if the dialog's file name is *CHILD.NS3*) to which the tool bar „belongs“ into the **Control ID** field of the **Attributes** window for the bar in the dialog editor. For each global bar, leave this field empty. If you wish the bar to be displayed only when no MDI child dialog is active, enter the name of the MDI frame dialog here.
4. For MDI applications, you should uncheck the **Enabled** check box in the **Attributes** window for each bar that should not be displayed by default.

```
DEFINE DATA
PARAMETER
  1 #DIALOG     HANDLE OF GUI
LOCAL
  1 #CONTROL       HANDLE OF GUI
  1 #ACTIVE-DLG    HANDLE OF GUI
  1 #CTXMENU       HANDLE OF CONTEXTMENU
  1 #MITEM-DYN     HANDLE OF MENUITEM
LOCAL USING NGULKEY1
END-DEFINE
*
DEFINE SUBROUTINE PROCESS-BAR-COMMANDS
  DECIDE ON FIRST *EVENT
    VALUE 'COMMAND-STATUS'
       PERFORM COMMAND-STATUS
    VALUE 'IDLE'
```

```
       PERFORM IDLE
    VALUE 'CLICK'
      PERFORM CLICK
    VALUE 'BEFORE-OPEN'
      PERFORM BEFORE-OPEN
    VALUE 'AFTER-OPEN'
      PERFORM AFTER-OPEN
   NONE
      IGNORE
   END-DECIDE
*
   DEFINE SUBROUTINE COMMAND-STATUS
     /* Must enable our commands, otherwise they're automatically disabled!
     #CTXMENU := #DIALOG.CONTEXT-MENU
     #MITEM-DYN := #CTXMENU.FIRST-CHILD
     REPEAT WHILE #MITEM-DYN <> NULL-HANDLE
        IF #MITEM-DYN.CLIENT-HANDLE <> NULL-HANDLE
          #MITEM-DYN.ENABLED := TRUE
        END-IF
        #MITEM-DYN := #MITEM-DYN.SUCCESSOR
     END-REPEAT
   END-SUBROUTINE
*
   DEFINE SUBROUTINE IDLE
     PERFORM SWITCH-BARS
   END-SUBROUTINE
*
   DEFINE SUBROUTINE CLICK
     #CONTROL := *CONTROL
     IF #CONTROL.TYPE = MENUITEM AND #CONTROL.PARENT = #DIALOG.CONTEXT-MENU
          #MITEM-DYN := #CONTROL
          #CONTROL := #MITEM-DYN.CLIENT-HANDLE
          IF #CONTROL <> NULL-HANDLE
             IF #MITEM-DYN.CHECKED = CHECKED
                 #CONTROL.ENABLED := FALSE
                 #CONTROL.VISIBLE := FALSE
         ELSE
                #CONTROL.ENABLED := TRUE
                #CONTROL.VISIBLE := TRUE
         END-IF
       END-IF
      END-IF
   END-SUBROUTINE
*
  DEFINE SUBROUTINE BEFORE-OPEN
     #CTXMENU := #DIALOG.CONTEXT-MENU
     #MITEM-DYN := #CTXMENU.FIRST-CHILD
     REPEAT WHILE #MITEM-DYN <> NULL-HANDLE
        IF #MITEM-DYN.CLIENT-HANDLE <> NULL-HANDLE
          #CONTROL := #MITEM-DYN.CLIENT-HANDLE
        IF #CONTROL.VISIBLE
          #MITEM-DYN.CHECKED := CHECKED
```

```
       ELSE
          #MITEM-DYN.CHECKED := UNCHECKED
       END-IF
     END-IF
     #MITEM-DYN := #MITEM-DYN.SUCCESSOR
    END-REPEAT
   END-SUBROUTINE
*
   DEFINE SUBROUTINE AFTER-OPEN
     /* for MDI frames, unsuppress IDLE event to track active child change
     IF #DIALOG.TYPE = MDIFRAME
       #DIALOG.SUPPRESS-IDLE-EVENT := NOT-SUPPRESSED
     END-IF
     /* if dialog has no context menu, create one
     #CTXMENU := #DIALOG.CONTEXT-MENU
     IF #CTXMENU = NULL-HANDLE
         PROCESS GUI ACTION ADD WITH PARAMETERS
            HANDLE-VARIABLE = #CTXMENU
            TYPE = CONTEXTMENU
            PARENT = #DIALOG
     END-PARAMETERS GIVING *ERROR
     #DIALOG.CONTEXT-MENU := #CTXMENU
   END-IF
   /* unsuppress context menu's BEFORE-OPEN event for item update
   #CTXMENU.SUPPRESS-BEFORE-OPEN-EVENT := NOT-SUPPRESSED
   /* display bars according to context
   PERFORM SWITCH-BARS
  END-SUBROUTINE
*
  DEFINE SUBROUTINE SWITCH-BARS
     IF #DIALOG.TYPE = MDIFRAME
       #ACTIVE-DLG := #DIALOG.ACTIVE-CHILD
     END-IF
     IF #ACTIVE-DLG = NULL-HANDLE
       #ACTIVE-DLG := #DIALOG
     END-IF
     IF #ACTIVE-DLG <> #DIALOG.CLIENT-HANDLE
       #CTXMENU := #DIALOG.CONTEXT-MENU
       IF #CTXMENU <> NULL-HANDLE
         /* Remove any dynamic menu items previously created
         #CONTROL := #CTXMENU.FIRST-CHILD
         REPEAT WHILE #CONTROL <> NULL-HANDLE
           #MITEM-DYN := #CONTROL.SUCCESSOR
           IF #CONTROL.CLIENT-HANDLE <> NULL-HANDLE
              PROCESS GUI ACTION DELETE WITH #CONTROL
           END-IF
           #CONTROL := #MITEM-DYN
         END-REPEAT
         /* Search for all tool bar and dialog bar controls
         #CONTROL := #DIALOG.FOLLOWS
         REPEAT WHILE #CONTROL <> #DIALOG
            IF #CONTROL.TYPE = TOOLBARCTRL OR
```

```
                #CONTROL.TYPE = DIALOGBAR
              #CONTROL.CLIENT-KEY := 'CONTROL-ID'
              IF #CONTROL.CLIENT-VALUE = ' ' OR
                  #CONTROL.CLIENT-VALUE = #ACTIVE-DLG.NAME
               #CONTROL.VISIBLE := #CONTROL.ENABLED
               /* Create menu entry for bar
               PROCESS GUI ACTION ADD WITH PARAMETERS
                 HANDLE-VARIABLE = #MITEM-DYN
                 TYPE = MENUITEM
                 PARENT = #CTXMENU
                 STRING = #CONTROL.STRING
                 SUCCESSOR = #MITEM-DYN
                 CLIENT-HANDLE = #CONTROL
               END-PARAMETERS GIVING *ERROR
             ELSE
               #CONTROL.VISIBLE := FALSE
             END-IF
           END-IF
           #CONTROL := #CONTROL.FOLLOWS
         END-REPEAT
       END-IF
       /* Save handle of currently active dialog
       #DIALOG.CLIENT-HANDLE := #ACTIVE-DLG
     END-IF
  END-SUBROUTINE
END-SUBROUTINE
END
```

96 Working with Error Events

When a runtime error occurs while a dialog is active, the dialog receives an error event. You can specify event-handler code to be executed whenever this error occurs. If no error event-handler code is specified, Natural aborts with an error message and all dialogs will be closed.

You can continue normal dialog processing after error handling by specifying an `ESCAPE ROUTINE` statement at the end of the event-handler code.

The dialog editor generates an `ON ERROR` statement for the event handler. If, for example, you want to prevent the end user from closing the entire application when trying to divide an integer by zero, and the parameter `ZD` is set to `ON`, the error event-handler code might look like this:

```
COMPRESS 'Natural error' *ERROR 'occurred.' INTO #DLG$WINDOW.STATUS-TEXT
    ESCAPE ROUTINE
```

97 Working with a Group of Radio Button Controls

Radio button controls are created just like push button controls or toggle button controls; however, they are grouped using the `GROUP-ID` attribute. If you define a number of radio button controls as a group, only one button is selected at any time. The `GROUP-ID` attribute provides this selection logic.

You group several radio button controls by assigning them the same `GROUP-ID` value (group number) in their attributes windows. If the end user clicks on a radio button control, all other radio-button controls in the dialog with the same `GROUP-ID` will be deselected. They will also be deselected if one radio button control is selected by code like the following:

```
...
   1 #RB-1 HANDLE OF RADIOBUTTON
...
#RB-1.CHECKED := CHECKED  /* Set the CHECKED attribute to value CHECKED
...
```

You also have to bear in mind that the end user should be able to use the keyboard for navigation inside a group of radio button controls: TAB selects the first radio button control, and the arrow keys enable you to navigate within the radio button group. To ensure that Natural automatically allows for such navigation, the radio button controls must follow each other directly in the navigation sequence. If you are dynamically adding a radio button control via the `PROCESS GUI` statement action `ADD`, this can be achieved by specifying a value for the button's `FOLLOWS` attribute.

To edit the navigation sequence

- From the **Dialog** menu, choose **Control Sequence**.

98 Working with Image List Controls

- Introduction 732
- Creating the Image List Control 732
- Adding Images 732
- Composite Images 733
- Scaling and Transparency 734
- Bitmaps vs. Icons 735
- Using an Image List 736
- Referencing Images from the Image List 736
- Overlay Images 737
- Modifying Images 738
- Deleting Images 739
- Deleting the Image List Control 739

Introduction

An image list control is a container of ordered images that can be associated with particular control types, such as list view and tree view controls. It allows images to be efficiently re-used by the control's items without the image being re-loaded from the disk each time. It also ensures that all images are compatible (e.g., are of the same size and color organization).

Creating the Image List Control

Image list controls are created, as usual, via the ADD action:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMGLST-1
   TYPE = IMAGELIST
   PARENT = #DLG$WINDOW
   STYLE = 'LS'
END-PARAMETERS GIVING *ERROR
```

An image list control may consist of up to two sets of images internally, one consisting of large images (typically 32 by 32 pixels) and one consisting of small images (typically 16 by 16 pixels). Which of these (if any) is created internally depends on the image list control's "Large Images (L)" and "Small Images (S)" STYLE flags. If neither of these flags are specified, a single set of images is created, with an explicit image size as determined by the image list control's ITEM-W and ITEM-H attribute values. If both of these are zero, small images are assumed.

Adding Images

Images are added to an image list by creating an image control, based on the required image (bitmap or icon) file, as a child of the image list control:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMG-1
   TYPE = IMAGE
   PARENT = #IMGLST-1
   BITMAP-FILE-NAME = 'example.bmp'
END-PARAMETERS GIVING *ERROR
```

Images are appended to the list by default, unless the SUCCESSOR attribute is used to insert them at a specific position.

Composite Images

Image controls can be categorized into two types: single-image image controls and multi-image image controls.

Single-image image controls contribute a single image to each set of images stored by the parent image list control. That is, if the image list contains both large and small images, one of each is provided by the image control. Single-image image controls may be bitmaps or icons.

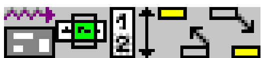
Multi-image image controls, as the name suggests, may contribute more than one image (in each required size) to the parent image list control. Multi-image image controls must be based on bitmap files, rather than icons, and are distinguishable from single-image image controls in that their "Composite image (C)" STYLE flag is set:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMG-1
   TYPE = IMAGE
   PARENT = #IMGLST-1
   STYLE = 'CsT'
   BITMAP-FILE-NAME = 'composite.bmp'
END-PARAMETERS GIVING *ERROR
```

The number of images in the composite bitmap is automatically calculated from the size of the bitmap and the width and height of the images in the (smallest) set of images stored by the parent image list control. Thus, in the case where both large and small images are stored, the bitmap would typically be 16 pixels high, and (16 * N) pixels wide, where N is the number of images to be stored in the image control. Here is an example of a composite bitmap containing five images:

Scaling and Transparency

In the example provided in the preceding section, two other style flags were specified in addition to the "Composite image (C)" style: namely, the "Scaled (s)" and the "Transparent (T)" style flags. The first of these is absolutely necessary if the parent image list control contains multiple sets of images in different sizes. For example, if large images are also being used, the flag causes the composite image to be scaled internally first before being chopped up into its constituent images, as follows:



Note that if the "Scaled (s)" style flag were not specified, the composite bitmap would be extended in the background color, rather than being scaled, before being chopped up, as shown below:

This would result in the following five large images:

Needless to say, this is not normally what you want!

The "Transparent (T)" style flag indicates that the images should be rendered transparently, such that all pixels in the bitmap's background color are not drawn. The background color can be explicitly specified by setting the `BACKGROUND-COLOUR-VALUE` and/or `BACKGROUND-COLOUR-NAME` attributes for the image control to the required value. Otherwise, if no color is specified (as in the previous example), the color of the first (i.e., top-left) pixel in the bitmap is taken as being the background color.

Of course, both the "Scaled (s)" and the "Transparent (T)" style flags can also be applied to non-composite images.

Bitmaps vs. Icons

Apart from not being able to source mulitple images (as described above), icons differ from bitmaps in two important ways. Firstly, a single icon (*.ICO*) file can contain multiple versions of the icons in different sizes. Thus when Natural requires the large image, and the source is an icon file, the large icon defined in the icon file is used, if present, in preference to synthesizing it from one of the other icons in the file by scaling. Similarly, when Natural requires the small image, and the source is an icon file, the small icon defined in the icon file is used, if present. In contrast, bitmap files do not contain multiple images, so if both large and small images are required for an image list, one of the two images (usually the large image) must by synthesized from the other as described in the previous section.

Secondly, icons typically contain a monochrome bitmap (known as the image mask) that determines which pixels in the image are transparent (i.e., should not be drawn). Thus, when Natural loads an image from an icon file, and the image control's `BACKGROUND-COLOUR-NAME` attribute is set to `DEFAULT` (or is not specified), and the image control's "Transparent (T)" style flag is specified without the "Scaled (s)" style flag, Natural uses the icon's transparency mask, instead of making the above-mentioned assumption that all pixels in the same color as the first pixel are to be rendered transparently, as is the case for images loaded from a bitmap file. If an explicit (i.e., non-default) background color is specified, all pixels in this color are treated as transparent, regardless of whether an icon or bitmap is being used. The icon's transparency mask is ignored here, as is also the case if the icon is scaled.

Therefore, if both large and small images are needed, it may be preferable to use single-image image controls based on icon files containing both large and small representations of the image, rather than use a multi-image image control based on a single composite bitmap. The use of individual icon (*.ICO*) files has the advantage that the and large representations of the image (assuming that both are provided in the file) can have different levels of detail. The main disadvantage is that it normally takes longer to load the images from multiple icon files than it does to load them from a single composite bitmap file.

Using an Image List

Before any images from the image list can be used by a control (such as the tree view or list view control), the image list must be associated with the control. This association is achieved by assigning the handle of the image list control to the host control's `IMAGE-LIST` attribute. For example:

```
#LV-1.IMAGE-LIST := #IMGLST-1
```

Having set the image list, the image list control's images are now available for use by the control's items.

Referencing Images from the Image List

To use a particular image from the parent control's image list for a particular item (e.g. list view item or tree view item), the image to be used has to be specified in one of two ways:

1. By setting the item's `IMAGE` attribute to the handle of the image control and (if necessary) the item's `IMAGE-INDEX` attribute to the relative offset of the required image (starting from zero) within the image control. If the image control only contains one image, it is not necessary to specify an image index. The image specified must belong to the image list control assigned to the item's container.
2. By setting the item's `IMAGE-INDEX` attribute to the ordinal of the image within the image list (1=first image, 2=second image, and so on). The item's `IMAGE` attribute must be either not specified or set to the default value of `NULL-HANDLE` in this case.

In the first case (relative indexing), wrap-around is used on the index. Thus, if an image control has N images, an image index of 0 refers to the first image in the image control, an image index of (N - 1) refers to the last image, and an image index of N refers to the first image again, and so on. Thus, if the image control only contains one image, the relative image index (if specified at all) has no effect: due to wrap-around, the first (and only) image will always be taken.

In the second case (absolute indexing), no wrap-around is used on the image index, which must be in the range 1 through to the number of images in the image list (inclusive). If the specified value is not in this range, no image is displayed for the specified item.

Note that the `IMAGE-INDEX` attribute can also be applied to an image control. In this case, the attribute is read-only, and returns the offset (starting from zero) of the image control's first image within the parent image list control.

One advantage of using relative indexing is that Natural keeps track the references to the specified image (both in the dialog editor and at run-time) and automatically propagates changes to the image control or to its position in the image list. In practice, absolute indexing is probably most

useful in situations where an image list control with a single composite (i.e., multi-image) image control is used, and where the images are not modified at run-time.

Overlay Images

There are situations where it is desirable to be able to offer several variations of an image. For example, the displayed image for an item representing a folder may need to be modified to indicated that the folder is active. Rather than providing an image of a folder and an image of an active folder, it may be more convenient to provide only the first of these images, and to indicate the active state via a second image containing only the „active“ symbol, which is then superimposed on the first. Such an image is referred to as an *overlay* image, to distinguish it from the underlying *base* image.

Overlay images are contained within the same image list that is used to display the base images, as determined via the host control's `IMAGE-LIST` attribute. They are therefore the same size as the base images, but are always rendered transparently, to allow the underlying image to show through.

To use an overlay image for an item, a value must be specified for the item's `OVERLAY` and/or `OVERLAY-INDEX` attributes. These attributes are used analogously to the `IMAGE` and `IMAGE-INDEX` attributes (respectively) for base images (see above).

For technical reasons, images intended for use as overlay images must be „pre-registered“. In Natural, this is done by setting the image list control's "O" (Overlay) `STYLE`. However, if the overlay controls are defined statically, this style is automatically set by the dialog editor. The presence or absence of this style distinguishes base images from overlay images. Consequently, the `OVERLAY` attribute (if specified) can only refer to an image control with this style, whereas the `IMAGE` attribute (if specified) can only refer to an image control without it. If absolute indexing (see above) is being used, the `IMAGE-INDEX` can refer to an overlay image (which is then „misused“ as a base image). However, a corresponding attempt to use the `OVERLAY-INDEX` attribute to refer to a base image fails (no overlay image is drawn).

Windows sets a limit on the number of overlay images that may be defined for an image list. This limit is currently 15. Note that if any composite overlay image controls are used, each sub-image in the composite bitmap counts separately towards this quota.

As an example, suppose we create an image control based on a composite image containing the individual overlay images, as follows:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMG-2
   TYPE = IMAGE
   PARENT = #IMGLST-1
   STYLE = 'COs'
   BITMAP-FILE-NAME = 'overlays.bmp'
END-PARAMETERS GIVING *ERROR
```

Then, we could create a list view item (say) using the second overlay image from the composite bitmap by executing the following code:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   TYPE = LISTVIEWITEM
   PARENT = #LV-1
   STRING = 'Item with overlay'
   IMAGE = #IMG-1
   IMAGE-INDEX = 3
   OVERLAY = #IMG-2
   OVERLAY-INDEX = 1
END-PARAMETERS GIVING *ERROR
```

In the above example, the list view item will use the fourth image from *COMPOSITE.BMP* as its base image, and the second image from *OVERLAYS.BMP* as the overlay image (relative image indexes are, as already mentioned, zero-based). Note that the list view item is created anonymously (i.e., no explicit `HANDLE-VARIABLE` attribute value specified).

Modifying Images

Image controls may be modified even if they are currently in use. For example:

```
#IMG-1.BITMAP-FILE-NAME := 'new.bmp'
```

Natural keeps track of, and automatically updates and redraws, each item that explicitly (i.e., via relative indexing) references an image from the modified image control. However, if absolute indexing is used, the corresponding items are not updated, even if they are implicitly referring to an image within the modified image control.

Deleting Images

Images may be removed from the image list by deleting the complete image control via the DELETE action. For example:

```
PROCESS GUI ACTION DELETE WITH #IMG-1 GIVING *ERROR
```

All items that *explicitly* (i.e., via relative indexing) reference an image from the deleted image control are automatically updated and redrawn to show no image.

However, if absolute indexing is being used, no automatic updating occurs. For example, suppose an image list control contains three single-image image controls and that items exists that refer to all three images via absolute indexing. If the second image control is deleted, the items that used to refer to the second image would suddenly reference the third image and the items that used to refer to the third image would „fall off the end“ and not reference anything. Furthermore, the controls containing the items would not automatically be redrawn to reflect the changes.

It is, of course, also possible to delete all images in the image list in one go, via the DELETE-CHILDREN action:

```
PROCESS GUI ACTION DELETE-CHILDREN WITH #IMGLST-1 GIVING *ERROR
```

This is equivalent to deleting each image in the image list individually.

Note that it is not possible to delete individual images within a composite (i.e., multi-image) image control.

Deleting the Image List Control

An image list control may be deleted when no longer required, even if it is in use. For example:

```
PROCESS GUI ACTION DELETE WITH #IMGLST-1 GIVING *ERROR
```

All controls using the image list control are updated accordingly, and their IMAGE-LIST attribute is automatically reset to NULL-HANDLE.

99 Working with List Box Controls and Selection Box Controls

List box controls and selection box controls contain a number of items. Both the controls and the items are dialog elements; the controls are the parents of the items.

There are two ways of creating list box items and selection box items:

- Use Natural code to create individual and multiple list box items dynamically; or
- use the dialog editor (to add single or arrays of list box items and selection box items).

In Natural code, this may look like this:

```
#AMOUNT := 5
ITEM (1) := 'BERLIN'
ITEM (2) := 'PARIS'
ITEM (3) := 'LONDON'
ITEM (4) := 'MILAN'
ITEM (5) := 'MADRID'
PROCESS GUI ACTION ADD-ITEMS WITH #LB-1 #AMOUNT #ITEM (1:5) GIVING #RESPONSE
```

You first specify the number of items you want to create, name the items, and use the `PROCESS GUI` statement action `ADD-ITEMS`.

If you want to go through all items of a list box control to find out which ones are selected, it is advisable to use the `SELECTED-SUCCESSOR` attribute because if a list box control contains a large number of items (100, for example), this helps improve performance. If you use `SELECTED-SUCCESSOR`, you have one query instead of 100 individual queries if you use the attributes `SELECTED` and `SUCCESSOR`.

Example:

```
/* Displays the STRING attribute of every SELECTED list-box item
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   .../* STRING display logic

    MOVE #LBITEM.SELECTED-SUCCESSOR TO #LBITEM
END-REPEAT
```

For performance reasons, you should not use the `SELECTED-SUCCESSOR` attribute to refer to the same dialog element handle twice, because Natural goes through the list of item handles twice:

```
/* Displays the STRING attribute of every SELECTED list-box item,
/* but may be slow
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   IF #LBITEM.SELECTED-SUCCESSOR = NULL-HANDLE /* Searches in the list of items
     IGNORE
   END-IF
   .../* STRING display logic
   MOVE #LBITEM.SELECTED-SUCCESSOR TO #LBITEM /* Searches in the list of items
END-REPEAT                                    /* for the second time
```

To avoid this problem, you use a second variable `#OLDITEM` besides `#LBITEM`:

```
/* Displays the STRING attribute of every SELECTED list-box item
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   #OLDITEM = #LBITEM
   #LBITEM = #LBITEM.SELECTED-SUCCESSOR/* Searches in the list of items (once)
   IF #LBITEM = NULL-HANDLE
     IGNORE
  END-IF
   .../* Display logic using #OLDITEM.STRING
END-REPEAT
```

If you retrieve the handle values of the selected items, a value other than NULL-HANDLE would normally be returned by selected items. Such a handle value can also be returned by non-selected items if you assign SELECTED-SUCCESSOR a value immediately before retrieving the SELECTED-SUCCESSOR value of a non-selected item, as shown in the following example:

```
...
PTR := #LB-1.SELECTED-SUCCESSOR
PTR := NOT_SELECTEDHANDLE.SELECTED-SUCCESSOR
IF NOT_SELECTEDHANDLE.SELECTED-SUCCESSOR = NULL-HANDLE THEN
   #DLG$WINDOW.STATUS-TEXT := 'NULL-HANDLE'
ELSE
   COMPRESS 'NEXT SELECTION: ' PTR.STRING TO #DLG$WINDOW.STATUS-TEXT
END-IF
...
```

If you want to query whether a particular item in a list box control is selected, you get the best performance by using the SELECTED attribute:

```
#DLG$WINDOW.STRING:= #LB-1-ITEMS.SELECTED(3)
```

Protecting Selection Box Controls and Input Field Controls

To prevent an end user from typing in input data in a selection box control or input field control, you have several possiblities, for example:

- setting the MODIFIABLE attribute to FALSE for the dialog element, or
- setting session parameter AD=P, or
- using a control variable (CV).

If a selection box control is protected, it is still possible to select items; only values from the item list will be displayed in its input field. If the STRING attribute is set to a value (dynamically or by initialisation) which is not in the item list, the value will not be visible to the end user.

100 Working with List View Controls

- Introduction 746
- View Modes 746
- Setting Item Images 748
- Item Placement 748
- Item Selection 750
- Item Activation 752
- List View Columns and Sub-items 752
- Sorting 755
- Label Editing 757
- Multiple Context Menus 759
- Drag and Drop 760

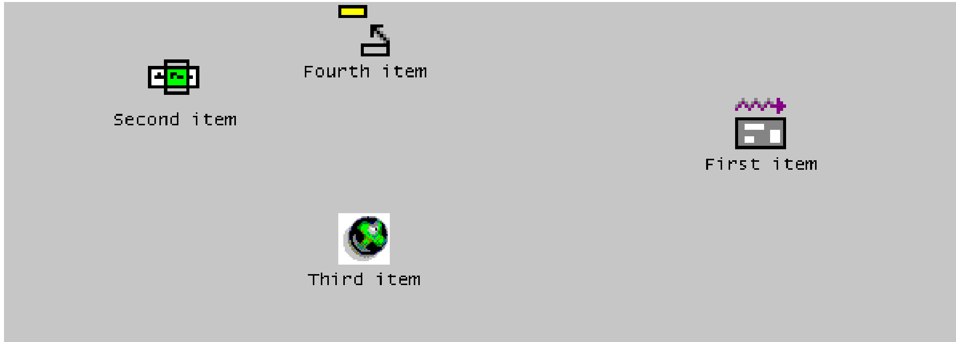
Introduction

A list view control can be used to display data in icon or column-based form. It is a very powerful control. Nevertheless, if you wish to display your data in tabular form and support direct in-place editing of any column value, you should consider using the table control instead.

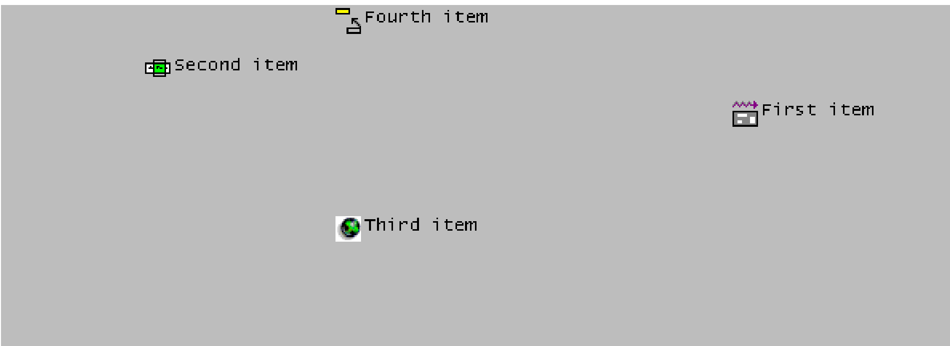
View Modes

List view controls in Natural can display their data in one of four view modes: icon, small icon, list or report.

In icon view mode, the data is displayed in large icon form:

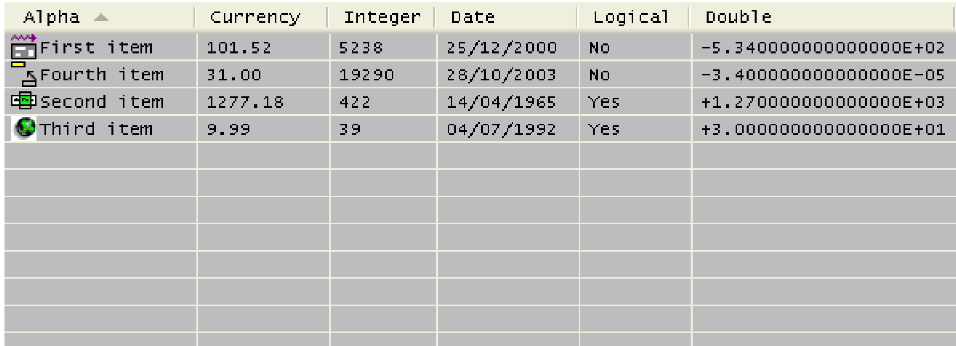


In small icon view mode, the item labels are displayed alongside the icons. As for the icon view, the items can optionally be displayed in arbitrary positions:



In list view mode, the items are displayed similarly to the small icon view, but cannot be freely positioned. Instead, they are displayed in columns:

In report view mode, each item occupies one row, and other data relating to the items may be displayed alongside the item in separate columns. A column header is also usually shown, as in the example below (although this can optionally be hidden by setting the list view control's "No header (x)" STYLE flag):



Alpha	Currency	Integer	Date	Logical	Double
First item	101.52	5238	25/12/2000	No	-5.340000000000000E+02
Fourth item	31.00	19290	28/10/2003	No	-3.400000000000000E-05
Second item	1277.18	422	14/04/1965	Yes	+1.270000000000000E+03
Third item	9.99	39	04/07/1992	Yes	+3.000000000000000E+01

In the report view, the columns can be resized by the user by dragging the column dividers. Alternatively, by double-clicking on the trailing column divider in the column header, the column's width is adjusted to fit the longest text within the column.

Note that the above example consists of eleven dialog elements: The list view control itself, four list view items and six list view columns. Both the list view items and the list view columns have the list view control as their PARENT, and are thus stored in the same SUCCESSOR chain. Although they can be interspersed, it is a good idea from both an organization and performance point-of-view to ensure that all list view columns precede all list view items. For example, if you are dynamically inserting a new column into a non-empty list view control as the last column, explicitly set its SUCCESSOR attribute to the handle of the first list view item, rather than not specifying it at all or setting it to NULL-HANDLE, which would cause the new column to be placed at the end of the chain, after all list view items.

The first list view column created for a list view control has a special significance, and is referred to (here) as the *primary* column. The primary column always displays the list view item labels (i.e.,

their STRING attribute values). The other columns display what is known as *sub-item* data. For example, the phrase „Currency sub-item" refers to the data stored in the "Currency" column (see above example). To refer to a particular value within the column, we would have to be more precise. For example, the value "1277.18" above could be referred to as the "Currency sub-item" for the "Second item" item. Sub-items are *not* dialog element types in Natural, and will be discussed in more detail below.

If no list view columns have been created, no information will be displayed in the list view control when it is in report view mode! However, it should be noted that the only way to switch between the view modes is programmatically by explicitly changing the list view control's VIEW-MODE attribute value. Therefore, if the application wishes to support multiple view modes, it must provide a mechanism (e.g., a context menu) for switching between them. So, in practice, the user should never see a list view control in report view mode that has no columns, since the application would normally not allow switching to this view mode in this case.

Setting Item Images

Images for the list view items may be defined by creating and associating an image list control with the list view control, then (for each item) selecting the required image from the image list via its index and/or image handle, as described in the section ***Working with Image List Controls***.

Please note that you should set the image list control's "Large images (L)" and "Small images (S)" styles according to the view modes that are to be supported. The icon view mode requires the availability of large images, whereas the other view modes require the availability of small images.

Item Placement

In the icon and small icon view modes, the list view items may be (re-)positioned by setting their RECTANGLE-X and/or RECTANGLE-Y attribute values. If no position is explicitly set on item creation, the items are laid out on an imaginary grid, with a default grid spacing that can be overridden by setting the list view control's SPACING-X and SPACING-Y attribute values. The ARRANGE action can be used at any time to either re-arrange the items to occupy consecutive locations based on this logical grid, or to snap the items to their nearest aligned logical grid position.

Note that in either of the two icon view modes, the list view item positions are interpreted as being in *view* coordinates, rather than being relative to the control's client area (as is the case in the other view modes). Unlike the client coordinates, the view coordinates of the items do not change when the icon view is scrolled. Conversion between view coordinates and client coordinates requires the use of the list view control's OFFSET-X and OFFSET-Y attributes, which return the origin of the client area in view coordinates.

Note that two list view control STYLE flags can override an explicit postion specified by the program. Firstly, if the control's "Auto-arrange (a)" style flag is specified, the items are automatically re-arranged on the imaginary grid each time an item is added or moved. In this case, an explicitly specified position merely indirectly determines the item's position in the arranged icon list. Secondly, if the control's "Snap to grid (r)" style flag is set, any item position explicitly specified by the program will be adjusted to the nearest aligned position on the imaginary grid. Note that this style is superfluous if the "auto-arrange" style is set.

Since users are often familiar with being able to modify list view item positions via drag and drop, it may be expected that the control automatically provides this capability. However, this is not the case. If the application wishes to support drag and drop, it must explicitly cater for it, as described in the next section.

In the list view mode, the items are always displayed in columns (as already mentioned above) and are not re-positionable. However, the spacing between adjacent columns may be set via the SPACING attribute.

Note that the list view control does not remember item positions when switching between view modes. For example, if you switch away from one of the icon view modes and then back to it again, the icons are always arranged. This behavior can be circumvented by providing explicit program code for saving and restoring item positions, as shown in the following example:

```
DEFINE SUBROUTINE SAVE-ITEM-POSITIONS
   #ITEM := #CONTROL.FIRST-CHILD
   REPEAT WHILE #ITEM <> NULL-HANDLE
     IF #ITEM.TYPE = LISTVIEWITEM
       #ITEM.CLIENT-KEY := 'RECTANGLE-X'
       #ITEM.CLIENT-VALUE := #ITEM.RECTANGLE-X
       #ITEM.CLIENT-KEY := 'RECTANGLE-Y'
       #ITEM.CLIENT-VALUE := #ITEM.RECTANGLE-Y
     END-IF
     #ITEM := #ITEM.SUCCESSOR
   END-REPEAT
END-SUBROUTINE
*
DEFINE SUBROUTINE RESTORE-ITEM-POSITIONS
   #ITEM := #CONTROL.FIRST-CHILD
   REPEAT WHILE #ITEM <> NULL-HANDLE
     IF #ITEM.TYPE = LISTVIEWITEM
       #ITEM.CLIENT-KEY := 'RECTANGLE-X'
       IF #ITEM.CLIENT-VALUE <> ' '
       #ITEM.RECTANGLE-X := VAL(#ITEM.CLIENT-VALUE)
       END-IF
       #ITEM.CLIENT-KEY := 'RECTANGLE-Y'
       IF #ITEM.CLIENT-VALUE <> ' '
         #ITEM.RECTANGLE-Y := VAL(#ITEM.CLIENT-VALUE)
       END-IF
     END-IF
```

```
     #ITEM := #ITEM.SUCCESSOR
   END-REPEAT
END-SUBROUTINE
*
DEFINE SUBROUTINE SWITCH-VIEW-MODE
   IF #VIEW-MODE <> #CONTROL.VIEW-MODE
     IF #CONTROL.VIEW-MODE = VM-ICON OR
        #CONTROL.VIEW-MODE = VM-SMALLICON
       PERFORM SAVE-ITEM-POSITIONS
     END-IF
     #CONTROL.VIEW-MODE := #VIEW-MODE
     IF #VIEW-MODE = VM-ICON OR
        #VIEW-MODE = VM-SMALLICON
       PERFORM RESTORE-ITEM-POSITIONS
     END-IF
   END-IF
END-SUBROUTINE
```

where the following local data definitions are assumed:

```
01 #CONTROL HANDLE OF GUI
01 #ITEM HANDLE OF GUI
01 #VIEW-MODE (I4)
```

The actual view mode switch can then be made by setting `#VIEW-MODE` to the desired view mode (one of the `VM-*` constants defined in the local data area NGULKEY1), setting `#CONTROL` to the handle of the list view control, and then calling the `SWITCH-VIEW-MODE` subroutine.

Item Selection

Items may be selected by the user either by clicking on them (optionally whilst holding down the CTRL key to perform an extended selection), or by defining a selection region by clicking within the list view control, holding down the primary mouse button, and dragging. The latter technique is known as marquee selection, and is only allowed if the control's "Marquee select (m)" `STYLE` flag is set (the default setting). Note that, if the control's "Hot-track select (t)" style flag is set, it is not necessary to click an item to select it. Instead, it is sufficient to simply let the mouse cursor hover over it briefly.

Alternatively, items may be selected or deselected programmatically by setting or clearing their `SELECTED` attribute.

In either case, extended selection is only available if the control's `MULTI-SELECTION` attribute is set to `TRUE`. Extended selection is the process of selecting new items, or deselecting old ones, without the existing selection being cleared first, and thus allows multiple (or no) items to be selected. In

the case of single selection list views, it is only possible for the user to implicitly deselect an item by selecting a new one. Marquee selection is also not available in this case.

The first (or only) selected item, if any, may be determined by querying the list view control's SELECTED-SUCCESSOR attribute, which returns NULL-HANDLE if there is no selection. The next selected item, if any, may be determined by querying a selected item's SELECTED-SUCCESSOR attribute. Iterative application of this technique allows complete enumeration of all selected items, as shown in the section below on drag and drop.

For each item that is selected or deselected, a CLICK event is raised for the list view control (if not suppressed), with the handle of the corresponding item being set in the control's ITEM attribute. Because many items may be selected in quick succession (e.g., via marquee selection), this event should not perform any lengthy processing. For example, it may be better to simply set a logical variable to TRUE in the CLICK event handler, indicating that more involved processing is required, and do the actual processing in the dialog's IDLE event handler in response to this flag being set. Don't forget to clear the flag after doing the work!

If the list view control's "Check boxes (c)" style flag is set, check boxes are displayed alongside each item. The item's CHECKED attribute may be used to retrieve or set an item's checked status programmatically. The first checked item, if any, may be determined by querying the list view control's CHECKED-SUCCESSOR attribute, and querying this attribute for a checked item returns the handle of the next checked item, if any, thus allowing complete enumeration of all checked items, as shown in the following example, which simply counts the number of checked items:

```
RESET #COUNT
#ITEM := #LV-1.CHECKED-SUCCESSOR
REPEAT WHILE #ITEM <> NULL-HANDLE
   ADD 1 TO #COUNT
   #ITEM := #ITEM.CHECKED-SUCCESSOR
END-REPEAT
```

where the following local data definitions are assumed:

```
01 #LV-1 HANDLE OF LISTVIEW
01 #ITEM HANDLE OF LISTVIEWITEM
01 #COUNT (I4)
```

Whenever an item is checked or unchecked, a CHECK event is raised for the list view control, if not suppressed via the SUPPRESS-CHECK-EVENT attribute, with the handle of the corresponding item being set in the control's ITEM attribute.

Item Activation

When a user double-clicks on an item, an ACTIVATE event is raised (unless suppressed) for the list view control. The application, if it decides to handle this event, normally performs a default action on each selected control. The default action is user-defined and can be different for each item. For example, activating an item representing a text file might cause the file to be opened in an editor, whereas activating an item representing an audio file might cause the file to be played. Note that there may be other, non-default, actions applicable to one or more of the selected items, but these are typically accessed via other mechanisms. For example, they may be listed (typically along with the default action) in a context menu displayed by the application.

If multiple selection is allowed (see above) and the CTRL key is held down whilst double-clicking an item, the selection state of the item is toggled before the ACTIVATE event is raised.

If either of the control's "Underline hot (u)" or "Underline cold (U)" STYLE flags are set, it is only necessary to single-click on an item in order to activate it.

The ACTIVATE event can also be triggered via the keyboard. This can be done in either of two ways:

1. By pressing the key or key combination defined for the list view control's ACCELERATOR attribute. The control need not currently have the focus.
2. By pressing the ENTER key, if the list view control currently has the focus. This method only works if the dialog neither contains a default pushbutton, nor a pushbutton with the "OK Button (O)" STYLE flag set.

In either case, no ACTIVATE event is raised if no items are currently selected.

List View Columns and Sub-items

Each column in a list view (as displayed when the list view control is in report view mode) can contain (at most) one item of data per list view item, which is then (if present) displayed for the item in that column. As already mentioned above, this data is known as sub-item data.

In order to be able to support sorting correctly (see next section), the sub-item data does not need to be alphanumeric, as it is per default, but can be one of any of the pre-defined types supported by the column's FORMAT attribute. In addition, an edit mask can be applied to the column by setting its EDIT-MASK attribute. The values seen in the report view column by the user for a column are the alphanumeric representations of the sub-item for that column, using the associated edit mask (if any). The conversion between the sub-item data and the displayed data is compatible with the MOVE EDITED statement if an edit mask is supplied. Otherwise, the conversion between the internal and displayed data, and vice-versa, is compatible with the conversion involved in copying the data to and from the Natural stack (see the STACK TOP DATA and INPUT statements). For example,

numeric values are displayed using the current decimal character (as defined by the DC parameter) if necessary, with a leading minus character if negative, date values are displayed in the format defined by the DTFORM parameter setting, logical values are displayed as an "X" if true and as a blank if false, and so on.

The first column defined for a list view control (the *primary* column) has a special significance: It always displays the item's label. Therefore, any changes to an item's sub-item data for the first coumn automatically update the item's label, and vice-versa. Otherwise, and for all other columns, the only means of updating the sub-item data is via the SET-SUBITEM-DATA action. When calling this action, the sub-item data must be supplied in a format compatible with the *internal* data type, as specified by the column's FORMAT attribute value (alphanumeric by default).

For example, suppose we add a column to a list view control as shown below:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #LVCOL-DATE
   TYPE = LISTVIEWCOLUMN
   STRING = 'Date'
   PARENT = #LV-1
   RECTANGLE-W = 83
   STYLE = 'l'
   FORMAT = FT-DATE
   EDIT-MASK = 'YYYY/MM/DD'
END-PARAMETERS GIVING *ERROR
```

The sub-item data for this column for a particular list view item, #LVITEM-1 (say), can then be set to the current date as follows:

```
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE *DATX GIVING *ERROR
```

Note that we could also have used *TIMX instead of *DATX, because time values in Natural are automatically convertible to date values. In either case, the value is then displayed as the current date in YYYY/MM/DD format. For the primary column, the display string is the item label, which means that the effects of the modification will be visible even if the list view control is not currently in report view mode.

Note also that the data is not supplied in display format. For example, the following will not work:

```
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE '2004/11/03' GIVING *ERROR  /* Does NOT work!
```

However, if the column happens to be the primary column, then it is alternatively possible to update the column data indirectly, by setting the item's label. For example:

```
#LVITEM-1.STRING := '2004/11/03'
```

Retrieval of the sub-item data may be achieved by calling the `GET-SUBITEM-DATA` action, which takes the same parameters as the `SET-SUBITEM-DATA` action. For example:

```
PROCESS GUI ACTION GET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE #DATE GIVING *ERROR
```

where `#DATE` is defined as follows:

```
01 #DATE (D)
```

One fact to bear in mind, however, is that there may be no sub-item data to be retrieved. For example, the sub-item data may not have been created, or may have been deleted (see below). Natural, however, does not support null values. Therefore, by default, Natural resets the receiving fields when a null value is returned (see the `RESET` statement). However, if a default value has been set for a list view column, this default value is returned instead. Setting a default value for a column is done by calling `SET-SUBITEM-DATA`, specifying `NULL-HANDLE` in place of a list box item handle. For example, for a numeric column, `#LVCOL-NUM`, where only positive values are allowed, we might choose to set the default value to -1, as shown below:

```
#NUM := -1
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  NULL-HANDLE #LVCOL-NUM #NUM GIVING *ERROR
```

where `#NUM` can be a field of any signed numeric format (e.g. I2).

The use of default values allows a value to be chosen by the programmer that does not match any explicit value that can be used in the program. If necessary, the program should be changed to prevent the default value being entered as explicit data.

For both the SET-SUBITEM-DATA and GET-SUBITEM-DATA actions, it is possible to set or get (respectively) the sub-item data (for a specific item) for multiple columns in a single statement. For example:

```
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-NUM #NUM #LVCOL-DATE #DATE GIVING *ERROR
```

In other words, multiple [column handle, receiving field] operand pairs may be specified.

To delete the subitem data (causing null values to be stored internally, as if no data had been set), use the DELETE-SUBITEM-DATA action. For example:

```
PROCESS GUI ACTION DELETE-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE GIVING *ERROR
```

Again, multiple sub-items may be deleted for a specific item in a single statement, by specifying multiple column handles:

```
PROCESS GUI ACTION DELETE-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE #LVCOL-NUM GIVING *ERROR
```

When list view items are deleted, their associated sub-item data (if any) is deleted with them. Note that if you wish to delete all items in a list view control, but leave the list view columns intact, use the CLEAR action:

```
PROCESS GUI ACTION CLEAR WITH #LV-1 GIVING *ERROR
```

where #LV-1 is the list view control handle.

Sorting

Sorting of the subitem data for a list view column may be achieved either by the user (if the list view control's "No header (x)" and "No sort header (y)" STYLE flags are not set), by clicking on a column header, or by the program, by calling the SORT-ITEMS action. In the latter case, items may be sorted even if no columns are available, by passing the handle of the list view control itself rather than the list view column. See the documentation for the SORT-ITEMS action for more information.

The sorting on clicking on a column in the column header of a list view control is normally implicitly performed by Natural. However, before performing the sort, Natural raises a `CLICK` event (if not suppressed) for the list view column. On returning from this event, Natural checks whether the column is already sorted in the required direction, and performs no further action if this is the case. This means that the application can perform the sort itself instead of Natural, as long as it obeys the rules for the sort direction (i.e., specifies descending sequence if the column is currently sorted in ascending sequence, or ascending sequence otherwise). This can be useful if the sort options (e.g. case-sensitivity) need to be dynamic, or if it is required to perform application-specific code after the sort. An example of a column `CLICK` event handler performing an explicit sort follows:

```
#CONTROL := *CONTROL
T1. SETTIME
IF #CONTROL.SORTED AND NOT #CONTROL.DESCENDING
  PROCESS GUI ACTION SORT-ITEMS WITH #CONTROL TRUE
    GIVING *ERROR
ELSE
  PROCESS GUI ACTION SORT-ITEMS WITH #CONTROL
    GIVING *ERROR
END-IF
COMPRESS 'Sort took' *TIMD(T1.) 'tenths of a second'
  INTO #DLG$WINDOW.STATUS-TEXT
```

However, in most cases, it will probably be sufficient to let Natural perform the sort implicitly.

For alphanumeric data, the sort column's "Case insensitive (i)" and "word compare (w)" `STYLE` flags determine the default way in which the values are compared. If the sort is done explicitly, the corresponding optional parameters to the `SORT-ITEMS` action, if specified, override these defaults. See the documentation for this action for more details on these options.

Missing („null“) values compare low. That is, they appear at the bottom of the column when the column is sorted in descending sequence, or at the top of the column when the column is sorted in ascending sequence. Furthermore, if two column entries are identical, the existing relative position of the two items concerned is preserved.

Note that, if the list view control is in one of the icon view modes, sorting the items causes them to be re-arranged. Therefore, if you are using explicit item positions in either of the icon view modes, it is probably a good idea to disable any sort commands.

List view controls also possess a `SORTED` attribute, implying that new items are inserted in their ascending or descending sort position, depending on the value of the control's `DESCENDING` attribute, rather than being inserted at the end of the item list. For this to work as expected, the items must already be sorted in the required direction. For example, if an item's label (i.e., its `STRING` attribute) is modified, the application itself should, if required, ensure that the list is maintained in sorted sequence. An example of how to do this is provided in the next section.

Note that the SORTED attribute does not influence the position of the items displayed in either of the icon views. However, if an explicit sort is performed via the SORT-ITEMS action, the items are re-arranged in the sorted sequence. If you cannot avoid doing a sort, and you are using explicit item positions in the icon view(s), then you must explicitly save the icon positions prior to the sort and restore them afterwards. For example:

```
PERFORM SAVE-ITEM-POSITIONS
PROCESS GUI ACTION SORT-ITEMS WITH #CONTROL GIVING *ERROR
IF #CONTROL.VIEW-MODE = VM-ICON OR
   #CONTROL.VIEW-MODE = VM-SMALLICON
  PERFORM RESTORE-ITEM-POSITIONS
END-IF
```

where #CONTROL is the handle of the list view control, and where the subroutines defined above for saving and restoring the item positions are used. Note that the icons are re-drawn (at their old positions), causing some flicker. Therefore, if possible, try to avoid performing a sort whilst the list view control is in one of the icon view modes. See the section below on label editing for an example of how this may be done.

Label Editing

The process of label editing for list view controls is the same as for tree view controls. Therefore, for more information on this subject, please refer to the section ***Label Editing in Tree View and List View Controls***.

Note that even if the SORTED attribute is set, the items are not automatically re-sequenced after a label editing operation has been completed. If this is required, this can be done as shown in the example below. Firstly, we define some variables for later use:

```
01 #SORTOBJ HANDLE OF GUI
01 #SORT (L)
01 #AUTO-ARRANGE (I4)
```

In addition, it is assumed that the list view control is named #LV-1.

In the list view control's AFTER-EDIT event, we cannot do the re-sequencing asynchronously, as this would interfere with the (as yet incomplete) editing process. Instead, we simply set the #SORT flag to indicate that the re-sequencing should occur at a later time, after the editing process has been completed:

```
#SORT := TRUE
```

In order to decide whether to perform a re-sequencing of the items, we will need to check whether the items are already sorted. We will do this by querying the SORTED and DESCENDING attributes of the primary column if the list view has columns, or those of the list view control itself otherwise. The relevant object handle is set in the dialog's AFTER-OPEN event:

```
IF #LV-1.COLUMN-COUNT <> 0
  #SORTOBJ := #LV-1.FIRST-CHILD
ELSE
  #SORTOBJ := #LV-1
END-IF
*
EXAMINE #LV-1.STYLE FOR 'a' GIVING NUMBER #AUTO-ARRANGE
IF #LV-1.SORTED AND #AUTO-ARRANGE <> 0 AND
  (#LV-1.VIEW-MODE = VM-ICON OR
   #LV-1.VIEW-MODE = VM-SMALLICON)
 #SORT := TRUE
END-IF
```

In addition, we obtain the status of the list view control's "Auto-arrange (a)" STYLE flag. If this is set, we can sort the items even if the control is in an icon mode, as we don't require the icon positions to be fixed. Furthermore, if the control is sorted, we indicate that we require the items to be initially re-sequenced. This is due to the fact (mentioned earlier) that the item positions are not updated on item insertion in the icon modes if the control's SORTED flag is set. The application thus needs to perform an initial sort itself in this case.

The actual work of re-sequencing the items is done asynchronously in the dialog's IDLE event:

```
IF #SORT
  IF #AUTO-ARRANGE <> 0 OR
    (#LV-1.VIEW-MODE <> VM-ICON AND
     #LV-1.VIEW-MODE <> VM-SMALLICON)
    IF #SORTOBJ.SORTED
     PROCESS GUI ACTION SORT-ITEMS WITH
       #SORTOBJ #SORTOBJ.DESCENDING GIVING *ERROR
    END-IF
    RESET #SORT
  END-IF
END-IF
```

Note that the sort is only done for the icon view modes for the list view control if it is auto-arranged. Otherwise, the #SORT flag is not reset, causing the re-sequencing to be deferred until the first IDLE event after the control is switched to a non-icon view mode.

Multiple Context Menus

If you wish to support just a single context menu for a control, you can simply set the control's CONTEXT-MENU attribute to the handle of the context menu you wish to display, and leave it set to this value. However, it is often required to be able to display more than one context menu for a particular control, whereby this approach is too inflexible.

To address the above problem, the CONTEXT-MENU event was introduced (not to be confused with the attribute of the same name as mentioned above!). This event (if not suppressed) is raised for the target control immediately before its CONTEXT-MENU attribute is evaluated, allowing the application to dynamically set this attribute to the handle of the appropriate context menu first.

As an example, assume that we have defined two context menus in the dialog editor: one containing item-related commands, #CTXMENU-ITEMS, and one containing generic commands (e.g., for switching the view mode for a list view control), #CTXMENU-DEFAULT. In this case, the following CONTEXT-MENU event could be used:

```
#CONTROL := *CONTROL
IF #CONTROL.SELECTED-SUCCESSOR <> NULL-HANDLE
   #CONTROL.CONTEXT-MENU:= #CTXMENU-ITEMS
ELSE
   #CONTROL.CONTEXT-MENU := #CTXMENU-DEFAULT
END-IF
```

where the following local data definition is assumed:

```
01 #CONTROL HANDLE OF GUI
```

In this example, the context menu #CTXMENU-ITEMS will be displayed if the user right clicks at a position occupied by an item, or #CTXMENU-DEFAULT otherwise.

Of course, this technique can be refined further to display context menus specific to the type(s) of the selected item(s).

Drag and Drop

Drag and drop may be used for re-positioning items within a list view control, as well as for data transfer to and from other windows. There is no difference between the two cases. The re-positioning scenario is merely a special case where the drop is made in the same list view control in which the drag was initiated.

The basic technique for providing drag and drop support is described in the section ***Using the Clipboard and Drag and Drop***. In particular, it should be noted that it is still necessary to place some data on the drag and drop clipboard even if it is only required to support re-positioning of the items within the control, in order to inform Natural that drag and drop should be initiated. Secondly, there is a caveat specific to list view controls, in that the re-positioning of items can change the origin of the list view control, so that the top near corner of the list view control's display area may no longer begin at the default origin of (0, 0).

The following example provides some code for demonstrating the use of drag and drop with the list view control, in order to support the following operations:

1. Re-positioning of list view items.
2. Dragging and dropping text from another application (e.g. WordPad) onto a list view item in order to change its label.

The first step is to ensure that the drag and drop modes are set correctly for the list view control. In the **List View Control Attributes** window in the dialog editor, set the **Drag mode** selection box to "Move" and the **Drop mode** selection box to "Copy+Move". This causes the control's `DRAG-MODE` and `DROP-MODE` attributes to be set to `DM-MOVE` and `DM-COPYMOVE`, respectively, in the generated source code for the dialog.

Next, the required local variables that are going to be used below must be defined:

```
01 #CONTROL HANDLE OF GUI
01 #DROP-ITEM HANDLE OF GUI
01 #ITEM HANDLE OF GUI
01 #SELECTED (L)
01 #AVAIL (L)
01 #X (I4)
01 #Y (I4)
01 #ORIG-X (I4)
01 #ORIG-Y (I4)
```

Having done this, we can write the necessary event handlers. The logical place to start is with the BEGIN-DRAG event:

```
#CONTROL := *CONTROL
*
PROCESS GUI ACTION INQ-CLICKPOSITION WITH
  #CONTROL #ORIG-X #ORIG-Y GIVING *ERROR
*
ADD #CONTROL.OFFSET-X TO #ORIG-X
ADD #CONTROL.OFFSET-Y TO #ORIG-Y
*
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH
  'DUMMYPRIVFMT' 0 GIVING *ERROR
```

This code consists of three parts:

1. Getting the click position in client coordinates.
2. Converting the click position to view coordinates (as mentioned above, the origin of the list view window is not always "(0, 0)").
3. Putting some dummy data on the drag and drop clipboard, otherwise Natural will not initiate the drag and drop operation. We choose a private clipboard format because, being only dummy data, we deliberately don't want other applications to recognize it.

Next, we provide a handler for the DRAG-ENTER event:

```
PROCESS GUI ACTION INQ-DRAG-DROP WITH
  1x #CONTROL GIVING *ERROR
*
IF #CONTROL = *CONTROL
  IF #CONTROL.VIEW-MODE = VM-ICON OR
     #CONTROL.VIEW-MODE = VM-SMALLICON
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := NOT-SUPPRESSED
  ELSE
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := SUPPRESSED
  END-IF
ELSE
  #CONTROL := *CONTROL
  PROCESS GUI ACTION INQ-FORMAT-AVAILABLE WITH CF-TEXT #AVAIL GIVING *ERROR
  IF #AVAIL
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := NOT-SUPPRESSED
  ELSE
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := SUPPRESSED
  END-IF
END-IF
```

The above code consists of three parts:

1. The INQ-DRAG-DROP action is called to determine the handle of the drag source control.
2. If the drag source is the current control, then drag source and drop target are identical. In other words, this is the item re-positioning case. Because item re-positioning is only allowed in one of the icon views, we unsuppress the DRAG-DROP event to allow the drop in this case. Otherwise, we suppress this event in order to prohibit a drop such that the „no drop“ drag cursor appears. Note that we could have also prevented a drag from occuring at all by not putting the data on the drag and drop clipboard in this case. However, although not demonstrated in this example, it is often desired to drag one or more items from a list view control to another window, even if the source control is in list or report view mode, for which the above code provides a better basis.
3. If the drag source and drop target are different, an attempt is being made to transfer data from another window. In this case, we check to see if data is available in the required format, CF-TEXT, and allow the drop if so. Otherwise the drop is prohibited.

To provide drop emphasis during the dragging of external data across the list view control, a DRAG-OVER event handler is supplied:

```
PROCESS GUI ACTION INQ-DRAG-DROP WITH
  1X #CONTROL 1X #X #Y GIVING *ERROR
*
IF #CONTROL <> *CONTROL
  #CONTROL := *CONTROL
  IF #CONTROL.SUPPRESS-DRAG-DROP-EVENT = NOT-SUPPRESSED
    PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
      #CONTROL #X #Y #ITEM GIVING *ERROR
    IF #ITEM <> #DROP-ITEM
      IF #DROP-ITEM <> NULL-HANDLE
        #DROP-ITEM.SELECTED := #SELECTED
      END-IF
      #DROP-ITEM := #ITEM
      IF #DROP-ITEM <> NULL-HANDLE
        #SELECTED := #DROP-ITEM.SELECTED
        #DROP-ITEM.SELECTED := TRUE
      END-IF
    END-IF
  END-IF
END-IF
```

The above code performs the following:

1. The INQ-DRAG-DROP action is called to determine the handle of the drag source control and the current drop position.
2. If the drag source and drag target are identical (item re-positioning), no further action is taken, as no drop emphasis is required in this case. Otherwise the INQ-ITEM-BY-POSITION action is used to find the list view item (if any) at the current drop position.

3. The current target item („drop item“) is tracked in #DROP-ITEM. Every time this changes, the current selection state of the new drop item is first checkpointed in #SELECTED, and then the item is selected to provide the drop emphasis by setting its SELECTED attribute to TRUE. In addition the selection state of the old drop item (if any) is restored to its previously checkpointed value.
4. Note that the drop emphasis is not applied if a drop is not possible (i.e., if the SUPPRESS-DRAG-DROP-EVENT attribute is set to SUPPRESSED).

To perform the actual drop, a DRAG-DROP event handler is supplied:

```
PROCESS GUI ACTION INQ-DRAG-DROP WITH
  1x #CONTROL 1x #X #Y GIVING *ERROR
*
IF #CONTROL = *CONTROL
  ADD #CONTROL.OFFSET-X TO #X
  ADD #CONTROL.OFFSET-Y TO #Y
  SUBTRACT #ORIG-X FROM #X
  SUBTRACT #ORIG-Y FROM #Y
  #ITEM := #CONTROL.SELECTED-SUCCESSOR
  REPEAT WHILE #ITEM <> NULL-HANDLE
    ADD #X TO #ITEM.RECTANGLE-X
    ADD #Y TO #ITEM.RECTANGLE-Y
    #ITEM := #ITEM.SELECTED-SUCCESSOR
  END-REPEAT
ELSE
  IF #DROP-ITEM <> NULL-HANDLE
    PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH CF-TEXT #DROP-ITEM.STRING
      GIVING *ERROR
    #DROP-ITEM.SELECTED := #SELECTED
    RESET #DROP-ITEM
  END-IF
END-IF
```

The above code performs the following:

1. The INQ-DRAG-DROP action is called to determine the handle of the drag source control and the current drop position.
2. The event handler differentiates between the case where the drag source and drop target controls are identical (item re-positioning) and the case where they are distinct (drag from external source).
3. In the item re-positioning case, the drop position is converted to view coordinates to obtain the "(x, y)"-displacement from the original position, which is then applied to each selected item to perform the move.
4. In the external source case, the text data is retrieved from the clipboard directly into the STRING attribute of the current drop item (if any) to update its label. It is not necessary to first check whether text is available on the drag and drop clipboard, as we did that in the DRAG-ENTER

event, prohibiting a drop and associated DRAG-DROP event from taking place at all if this is not the case. After updating the label, the drop item's previous selection state (prior to the drag) is restored and #DROP-ITEM reset ready for the next drag operation (if any).

Lastly, in the case that the user cancels the drag operation, or exits the bounds of the list view control without having performed a drop, a DRAG-LEAVE event handler is supplied:

```
IF #DROP-ITEM <> NULL-HANDLE
  #DROP-ITEM.SELECTED := #SELECTED
  RESET #DROP-ITEM
END-IF
```

The above code simply clears the drop emphasis (if any), by restoring the old drop item selection state. To satisfy the logic provided for the other event handlers, #DROP-ITEM is reset, indicating the absence of a drop item. This is essentially the same code as performed at the end of the DRAG-DROP event, since the DRAG-LEAVE event is not called in the case of a drop. Therefore, any drop emhasis resetting needs to be done in both places.

101 Working with Nested Controls

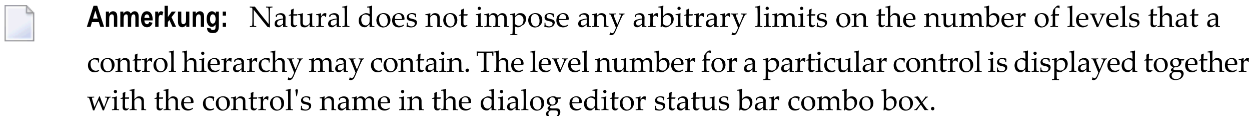
- Introduction 766
- Which Control Types can be Containers? 767
- Creating a Nested Control 767
- Multiple Selection, Control Sequence and Clipboard Operations 768

Introduction

It is possible to create controls as children of other controls in addition to so-called „top-level“ controls, which are direct children of the dialog. Such controls are referred to as nested controls. The parent control is referred to as the container. We will also use the term siblings to refer to a set of child controls which all have the same parent. Clearly, there can be many different sets of sibling controls within a control hierarchy.

Creation of a control hierarchy enables the Natural programmer to group together controls such that they can be manipulated more easily and more efficiently within a Natural program. The following list describes the characteristics of nested controls:

- Their position is relative to the client area of the container control instead of relative to the dialog.
- Their display is clipped to their respective ancestor windows. This means that the areas of the nested control that are outside the boundary of its container are not visible. The dialog editor does not allow dragging of nested controls outside of the container.
- Nested controls are always displayed in front of their container control, regardless of their position in the control sequence.
- Nested controls are moved with their container control. This applies at both edit-time in the dialog editor (when the container is dragged) and at runtime (when the container's `RECTANGLE-X` and/or `RECTANGLE-Y` attributes are modified).
- Nested controls are hidden when the container control is hidden, even though the `VISIBLE` attribute of the nested control remains unchanged.
- Nested controls are disabled at runtime when the container control is disabled, even though the `ENABLED` attribute of the nested control remains unchanged and even though the control does not become grayed.
- Nested controls are deleted when the container control is deleted.



Anmerkung: Natural does not impose any arbitrary limits on the number of levels that a control hierarchy may contain. The level number for a particular control is displayed together with the control's name in the dialog editor status bar combo box.

Which Control Types can be Containers?

Not all control types are capable of acting as a container. It is not possible to create a control as a child of an input field, for example. There are currently three types of container control supported by Natural:

- Group frames that have the (new) "container" style set. This can be changed in the dialog editor (via its attributes window) after the group frame has been created. If a group frame is converted to a container, all controls that are spatially contained within it are moved in the control hierarchy to become descendants of the group frame. If a group frame is converted to a non-container, all direct children of the group frame are moved up a level in the hierarchy to become siblings of the group frame.
- ActiveX controls which are marked as "OLEMISC_SIMPLEFRAME" in the registry. This flag is fixed by design for a particular ActiveX control class.
- Control boxes. This control type is always a control container. Indeed, that is its entire purpose in life. See the section ***Working with Control Boxes*** for more information.

Creating a Nested Control

Nested controls are created in the dialog editor in the same way as non-nested controls are. If, during control insertion, the initial left mouse button click is determined to be over a container control, the new control is created automatically by Natural as a child of that container. Even before the mouse button is clicked in insert mode, the dialog editor's status bar is continually updated with the container-relative mouse co-ordinates as the mouse cursor traverses the dialog.

In addition, nested controls can be indirectly created within the dialog editor when converting group frames to containers as described above.

At runtime, nested controls can be created dynamically, via the `PROCESS GUI ACTION ADD` statement for the nested control, by specifying the `PARENT` attribute as the handle of the required container control instead of the handle of the dialog. The nested control's position (`RECTANGLE-X` and `RECTANGLE-Y` attributes) should be specified relative to the container's client area. The client area of a control is the internal area of a control, excluding frame components such as 3-D borders, single-pixel frames resulting from use of the "Framed" style, and a control's scroll bars.

Multiple Selection, Control Sequence and Clipboard Operations

The dialog editor prohibits selection of multiple controls which do not have the same parent (i.e., are not all siblings of each other). This applies regardless of whether multiple controls are selected via „rubber banding“ (marking of a region with the left mouse button held down) or via extended selection (holding down the SHIFT key whilst selecting a control). However, if a selected container control is deleted, then all its direct and indirect children (*descendants*) are of course implicitly deleted also, even though they are not explicitly selected. For this reason, a clipboard cut operation always copies the selected control(s) *and* all descendant controls (if any) to the clipboard. For a clipboard copy operation, it is not clear whether to copy the container alone, or the container plus all its descendants. In this case, a message box is displayed, allowing the user to choose between the two options.

The pasting of controls from the clipboard uses the same control sequence (tab order) insertion position logic as for a control created from scratch. In both cases, the new control is created at a position in the control sequence immediately following the selected sibling (if any) plus any of its successive descendants. If a control other than a sibling is selected, an „effective sibling“ is used instead, based on the position of the (active) selected control in the control sequence. The „active“ selected control is the selected control (if any) which is highlighted using black (rather than gray) selection handles. If no selection is active, the control is inserted into the control sequence immediately preceding the first sibling control, or immediately after its container (or at the front of the control sequence for top-level controls) if the container is empty. Note, however, that the control sequence is maintained independently of the hierarchy. After a control has been created, it is possible to explicitly move any control to any position in the control sequence via the **Control Sequence** command on the **Dialog** menu.

The position of the newly-created control in the hierarchy is determined slightly differently in these two cases. In the case of a control being created from scratch, the container is determined by searching for the (topmost) container at the position where the left mouse button was pressed. However, in the case of pasting from the clipboard, we have no "(X, Y)"-position which we can use. In this case, the container is assumed to be the container of the selected control(s), or the dialog itself if no controls are selected. This means that if, for example, it is desired to copy and paste a control from one container to another, a control within the second container must be selected prior to the paste, not the container itself. If the second container is empty, this requires temporary creation of a dummy child control first, which can be deleted after the paste operation is complete.

Deletion of controls also deletes any of their descendant controls.

With the introduction of nested controls, the **Select All** command has been changed to operate in the following manner:

- If no control is currently selected, the command selects all top-level controls.
- Otherwise, all other controls that are siblings of the currently selected one(s) are additionally selected.

Thus, in the common case where only one level of hierarchy is in use, the **Select All** command continues, as before, to select all dialog controls.

102 Working with a Dynamic Information Line

Event-driven applications are much more user-friendly when text in the dynamic information line (DIL) explains the dialog element that currently has the focus. A dialog element has the focus if it can receive the end user's keyboard input.

You have two options to relate a dialog element to a DIL text:

- Use the dialog editor (most likely because it is the easiest way); or
- use Natural code to specify everything dynamically.

When you use the dialog editor, you will have to go through the following steps:

1. Set the attribute `HAS-DIL` to `TRUE` for the dialog by marking the **Dyn. Info Line** entry in the **Dialog Attributes** window.
2. Set the attribute `DIL-TEXT` to "*diltextstring*" for the dialog element. Choose the **Source...** button to the right of the **DIL Text:** entry in the attributes window. The window **Specify attribute Source** appears. Choose one of the attribute sources and enter the text in the **Value** field. Ensure that "*diltextstring*" explains the dialog element's usage in a short phrase.

When you use Natural code, the above two steps may look like this:

```
...
PERSDATA-DIALOG.HAS-DIL := TRUE  /* Set HAS-DIL To TRUE
#PB-1.DIL-TEXT := 'DILTEXTSTRING'   /* Assign the text string
...
```

Anmerkung: The `STATUS-TEXT` and the `DIL-TEXT` are displayed in the same area if the dialog has a status line and a text is displayed on the DIL.

103 Working with Spin Controls

- Introduction 774
- Up-Down Control 774
- Buddy Control 774
- Date and Time Formats 775
- Inputting Dates and Times 776
- Null Values 777
- Calendar Colors and Font 777

Introduction

A spin control consists of a pair of vertically-opposed arrow buttons known as an „up-down" control, optionally with an associated input field control known as a „buddy" control. The spin control has an associated integer range and current position. The buddy control (if present) displays the contents relating to the current position. The current position may be changed either by using the arrow buttons, the up and down cursor keys, or by typing the value directly into the buddy input field (if available and modifiable).

Up-Down Control

The up-down control allows the user to explicitly scroll through the spin control's integer range. The up-down control's buttons can also be implicitly selected by using the up arrow and down arrow cursor keys. Holding down the button or key causes the value to be incremented or decremented repeatedly, cycling round to the opposite end of the range when the corresponding limit is reached, if the control's "Wrap (w)" style is set. Initially, the incrementation or decrementation step is 1, but this increases after a few seconds. This acceleration may be disabled or modified via the `SET-ACCELERATION` action.

The control's range may be defined by setting its `MIN` and `MAX` attributes. The control's current position within this range may be programmatically set or obtained by setting or querying the `POSITION` attribute value, respectively.

Whenever the current position is changed by the user, a `CHANGE` event is raised for the control (if not suppressed). This event is not raise if the control's value is changed programatically. Alternatively, if a buddy input field control is present, the `CHANGE` event of the buddy control can be used.

Buddy Control

If the control's "Left align (l)" or "Right align (r)" style flag is set, the spin control contains an input field, known as the buddy control, which on the right or left of the up-down control respectively (the alignment relates to the up-down control, not to the buddy control).

The buddy control is a child of the spin control, and appears as a standard Natural input field control. This gives the Natural programmer access to all the features available for standalone input field controls. For example, the buddy control can be made to accept only digits and/or be made non-modifiable, for example.

If the spin control's "Set buddy (s)" style is set, the buddy control is automatically updated with the current position of the up-down control when the current position is changed. Otherwise, the

contents of the buddy control mus be updated manually in response to the spin control's `CHANGE` event.

Date and Time Formats

By default, the date and time information is displayed according to the date and time formats defined for the current regional settings. Because Windows provides two alternative date formats, one long and one short (both of which may be changed by the user), and because the short date format may not contain century information, one of three `STYLE` flags determines which of the standard date formats should be used. These (mutually exclusive) formats are:

- "Short date (s)", implying that the standard short date format for the current regional settings should be used.
- "Century date (c)", implying that the standard short date format for the current regional settings should be used, but extended to provide century information if this is not already the case. Note that in many cases, the short date format already includes century information, in which case this style does not change the appearance of the date.
- "Long date (d)", implying that the standard long date format for the current regional settings should be used.

In addition, the "Time (t)" style flag is provided in order to indicate that the control should display time (instead of date) information.

If these standard formats are not sufficient, they can be overridden by proving a custom format string using the `EDIT-MASK` attribute. Note, however, that the format string specifiers do not correspond to those used for edit masks elsewhere within Natural. The following table lists the available specifiers and their meanings:

Specifier	Description
d	The one- or two-digit day.
dd	The two-digit day. Single-digit day values are preceded by a zero.
ddd	The three-character weekday abbreviation.
dddd	The full weekday name.
h	The one- or two-digit hour in 12-hour format.
hh	The two-digit hour in 12-hour format. Single-digit values are preceded by a zero.
H	The one- or two-digit hour in 24-hour format.
HH	The two-digit hour in 24-hour format. Single-digit values are preceded by a zero.
m	The one- or two-digit minute.
mm	The two-digit minute. Single-digit values are preceded by a zero.
s	The one- or two-digit second.

Specifier	Description
ss	The two-digit second. Single-digit values are preceded by a zero.
M	The one- or two-digit month number.
MM	The two-digit month number. Single-digit values are preceded by a zero.
MMM	The three-character month abbreviation.
MMMM	The full month name.
t	The one-letter AM/PM abbreviation (that is, AM is displayed as "A").
tt	The two-letter AM/PM abbreviation (that is, AM is displayed as "AM").
yy	The last two digits of the year (that is, 2005 would be displayed as "05").
yyyy	The full year (that is, 2005 would be displayed as "2005").

In addition, any characters in quotes are displayed exactly as specified. To specify the quote character itself within a quoted string, two consecutive single quote characters should be used. Spaces and punctuation marks (such as the comma) do not need to be quoted.

For example, in order to display the string "John's birthday is Friday, December 31, 1969", the DTP control's `EDIT-MASK` attribute would be set to "'John''s birthday is' dddd, MMMM d, yyyy".

Inputting Dates and Times

The DTP control provides several ways of modifying the specified information:

- By the user, by entering numerical information (day numbers, etc.) directly.
- By the user, by incrementing or decrementing the selected field (e.g. day number, month name) via the + or - keys, respectively.
- By the user, if the DTP control has either the "Time (t)" or "Up-down (u)" style, by selecting the required field and incrementing or decrementing the value via the up-down („spin“) control.
- By the user, if the DTP control is using a month calendar, by pressing the down arrow to open the month calendar and navigating to the required date. Unlike the above method, this method updates all date fields simultaneously.
- Programmatically, by updating the `TIME` attribute with the required date or time.

For example, to set the date or time in a DTP control to the current date or time, use the following assignments:

```
#DTP-1.TIME := *DATX
```

or

```
#DTP-1.TIME := *TIMX
```

respectively, where `#DTP-1` is assumed to be the handle of the DTP control.

Note that the DTP control stores both date and time information, even though it only allows editing of the date or time component, depending on the control's style.

Null Values

If the "Allow 'no value' (n)" style is specified for the DTP control, the control displays a check box. If this check box is unchecked, the interpretation is that there is no date or time associated with the control. The application can test for this state by querying the control's `CHECKED` attribute. It can also revert the control to the „no value“ state by setting the `CHECKED` attribute back to `UNCHECKED`. Note that it is, however, not possible to explicitly set the `CHECKED` attribute to `CHECKED`, as this is done implicitly whenever a date or time is applied to the control. Furthermore, the `CHECKED` attribute may not be set at all for DTP controls without the "Allow 'no value' (n)" style.

Calendar Colors and Font

The colors and font used by the month calendar (if any) associated with the DTP control may be changed by use of the `SET-AUX-COLOR` and `SET-AUX-FONT` actions, respectively.

104 Working with a Status Bar

In a similar way as the dynamic information line, the status bar makes an event-driven application more user-friendly.

The programmer has two options to relate a dialog element to a status bar:

- use the dialog editor (most likely because it is the easiest way); and
- use Natural code to specify everything dynamically.

When you use the dialog editor, you will have to:

- Set the attribute `HAS-STATUS-BAR` to `TRUE` for the dialog by marking the **Status Bar** entry in the **Dialog Attributes** window. The `HAS-STATUS-BAR` attribute determines whether the status bar may be modified. If `HAS-STATUS-BAR` is false, but `HAS-DIL` is true, the status bar appears, but is only used as dynamic information line.

When you use Natural code, the above step may look like this:

```
...
PERSDATA-DIALOG.HAS-STATUS-BAR := TRUE  /* Set HAS-STATUS-BAR To TRUE
PERSADTA-DIALOG.STATUS-TEXT := 'HELLO'  /* Set the text to 'Hello'
...
```

Anmerkung: The `STATUS-TEXT` and the `DIL-TEXT` are displayed in the same area if the dialog has a status line and a text is displayed on the DIL.

105 Working with Status Bar Controls

- Introduction 782
- Creating a Status Bar Control 782
- Using Status Bar Controls without Panes 782
- Outputting Text to a Status Bar Control 783
- Sharing a Status Bar in an MDI Application 784
- Pane-specific Context Menus 785

Introduction

Anmerkung: Status bar controls are not to be confused with the traditional dialog status bar which is created by selecting the **status bar** check box in the **Dialog Attributes** window in the dialog editor, or by setting the dialog's `HAS-STATUS-BAR` attribute at run-time. If you are using status bar controls, you should leave the **status bar** check box unchecked and not set the `HAS-STATUS-BAR` attribute.

Creating a Status Bar Control

Status bar controls are created in the dialog editor in the same way as other standard controls (such as list boxes or push buttons) are. That is, they are either created statically in the dialog editor via the **Insert** menu or by drag and drop from the Insert tool bar, or dynamically at run-time by using a `PROCESS GUI ACTION ADD` statement with the `TYPE` attribute set to `STATUSBARCTRL`.

Unlike most other control types, status bar controls cannot be nested within another control and cannot be created within an MDI child dialog. In an MDI application, the status bar control(s) must belong to the MDI frame dialog.

A status bar control may have zero or more panes associated with it. Panes may be defined in the dialog editor from within the status bar control's attribute window, or at run-time by performing a `PROCESS GUI ACTION ADD` statement with the `TYPE` attribute set to `STATUSBARPANE`.

Using Status Bar Controls without Panes

A status bar control without panes offers restricted functionality, because most attributes providing access to the enhanced functionality of status bar controls are only supported for status bar panes. If you wish to do more with a status bar control than simply display a line of text, but don't need to split up the status bar control into multiple sections, you should create a single pane that occupies the full width of the status bar control.

Stretchy vs. non-stretchy panes

If panes are defined for a status bar control, it should be decided whether each pane should stretch (or contract) when the containing dialog is resized, or whether it should maintain a constant width. The former are referred to here as „stretchy“ panes, and the latter as „non-stretchy“ panes.

There is no explicit flag in the **Status Bar Control Attributes** window to mark a pane as stretchy or non-stretchy. Instead, any pane defined with a width (`RECTANGLE-W` attribute) of 0 is implicitly assumed to be a stretchy pane, whereas any panes with a non-zero width definition are implicitly assumed to be fixed-width panes of the specified width (in pixels). Because the `RECTANGLE-W` attribute defaults to 0, all panes are initially stetchy when defined in the dialog editor.

The width of a visible stretchy pane is determined by taking the total width available for all panes in the status bar control, subtracting the widths of all visible fixed-width panes, then dividing the result by the number of visible stretchy panes.

Anmerkung: The total available width for all panes normally excludes the sizing gripper, implying that the last pane stops short of the gripper, if present. However, if the status bar control has exactly one pane, and that pane is a stretchy pane, the full width of the dialog (including any sizing gripper) is used.

Outputting Text to a Status Bar Control

Text can be output to the status bar control in one of three ways:

1. For status bar controls with panes, by setting the `STRING` attribute of the pane whose text is to be set.
2. By setting the `STRING` attribute of the status bar control itself, which is equivalent to setting the `STRING` attribute of the first stretchy pane (if any) for status bar controls with panes.
3. By setting the `STATUS-TEXT` attribute of the dialog. This is equivalent to setting the `STRING` attribute of the status bar control (if any) identified by the dialog's `STATUS-HANDLE` attribute.

Note that the last method is often the most convenient for setting the message text, because it does not require a knowledge of the status bar control or pane handles.

Example:

```
DEFINE DATA LOCAL
01 #DLG$WINDOW  HANDLE OF WINDOW
01 #STAT-1      HANDLE OF STATUSBARCTRL
01 #PANE-1      HANDLE OF STATUSBARPANE
END-DEFINE
...
#DLG$WINDOW.STATUS-HANDLE := #STAT-1
...
#PANE-1.STRING := 'Method 1'
...
#STAT-1.STRING := 'Method 2'
...
#DLG$WINDOW.STATUS-TEXT := 'Method 3'
```

Anmerkung: The dialog editor automatically generates code to set the STATUS-HANDLE attribute to the first status bar control (if any). Therefore, the STATUS-HANDLE attribute only needs to be set explicitly if you are dynamically creating status bar controls, or if you have defined more than one status bar control in a dialog, and wish to switch between them.

Sharing a Status Bar in an MDI Application

Because status bar controls cannot be created for MDI child dialogs, it is convenient to not have to define multiple status bar controls in the MDI frame dialog. An alternative method is to define just a single status bar, and share it between each child dialog. This can be achieved as follows:

1. Define all possible panes you wish to use in your application within a single status bar control in the MDI frame dialog.
2. Mark all panes as "shared".
3. Export the handles of all panes to corresponding shadow variables in a GDA, so that the MDI child dialogs can access them directly.
4. In the COMMAND-STATUS event handler, set the VISIBLE attribute of all panes you wish to display for that dialog to TRUE. All other panes will be automatically made invisible.

Anmerkung: In the COMMAND-STATUS event, you must also set the ENABLED state of any commands (signals, or menu or tool bar items which do not reference another object via their SAME-AS attribute) associated with the dialog, otherwise they will be automatically disabled. The commands associated with the dialog are all non-shared commands for the MDI frame and all shared commands for the active MDI child (or MDI frame, if no MDI child dialog is active).

Pane-specific Context Menus

Context menus are defined for the status bar control and not per-pane. However, if you wish to ensure that the context menu for a status bar control only appears when the user right clicks a particular pane, you can associate a context menu with the status bar control, but suppress it if the user clicks outside that pane.

Example:

```
DEFINE DATA LOCAL
01 #CTXMENU-1   HANDLE OF CONTEXTMENU
01 #STAT-1      HANDLE OF STATUSBARCTRL
01 #PANE-1      HANDLE OF STATUSBARPANE
01 #PANE-2      HANDLE OF STATUSBARPANE
01 #PANE-3      HANDLE OF STATUSBARPANE
01 #PANE        HANDLE OF STATUSBARPANE
01 #X (I4)
01 #Y (I4)
END-DEFINE
...
#STAT-1.CONTEXT-MENU := #CTXMENU-1
...
DECIDE ON FIRST *CONTROL
   ...
   VALUE #CTXMENU-1
       DECIDE ON FIRST *EVENT
           ...
             VALUE 'BEFORE-OPEN'
             /* Get click position relative to status bar control
             PROCESS GUI ACTION INQ-CLICKPOSITION WITH
               #STAT-1 #X #Y GIVING *ERROR
             /* Get pane (if any) at specified position
             PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
               #STAT-1 #X #Y #PANE
             /* Only show context menu if user clicked in second pane
             IF #PANE = #PANE-2
                #CTXMENU-1.ENABLED := TRUE
             ELSE
                #CTXMENU-1.ENABLED := FALSE
             END-IF
        ...
    END-DECIDE
 ...
END-DECIDE
...
END
```

Anmerkung: If you wish to display a different context menu for different status bar panes, the menu items must be created dynamically in the context menu's `BEFORE-OPEN` event.

106 Working with Tab Controls

- Creating a Tab Control 788
- Assigning Controls to Tabs 788
- Use of Control Boxes as Tab Control Pages 789
- Switching Between Controls Belonging To Different Tabs 790
- Mixing Tab-dependent and Tab-independent Controls 791
- Keyboard Navigation 791
- Tab Switching Events 792

Creating a Tab Control

Tab controls are created in the dialog editor in the same way as other standard controls (such as list boxes or push buttons) are. That is, they are either created statically in the dialog editor via the **Insert** menu or by drag and drop from the Insert tool bar, or dynamically at run-time by using a `PROCESS GUI ACTION ADD` statement with the `TYPE` attribute set to `TABCTRL`.

Alternatively, dialogs containing tab controls may be generated with the Dialog Wizard. In this case, many of the techniques described in this section are applied automatically by the wizard, and either do not need to be explicitly implemented, or simply need to be extended or „filled-out“, whilst retaining the generated structure. This can significantly reduce the programming effort required.

A tab control may have zero or more tabs associated with it. Tabs may be defined in the dialog editor from within the tab control's attribute window, or at run-time by performing a `PROCESS GUI ACTION ADD` statement with the `TYPE` attribute set to `TABCTRLTAB`.

Assigning Controls to Tabs

The tab control is a container. However, the individual tabs are not containers, in the Natural implementation of this control. When controls are created within the tab control in the dialog editor, the control's `PARENT` attribute is automatically set to the handle of the tab control, and not to the handle of the currently active tab (if any). In order to associate a child control with a particular tab, the child control's `OWNER` attribute is set to the handle of the tab with which the control should be associated. The control is then automatically hidden by Natural when the tab is deactivated, and automatically re-shown when it is re-activated.

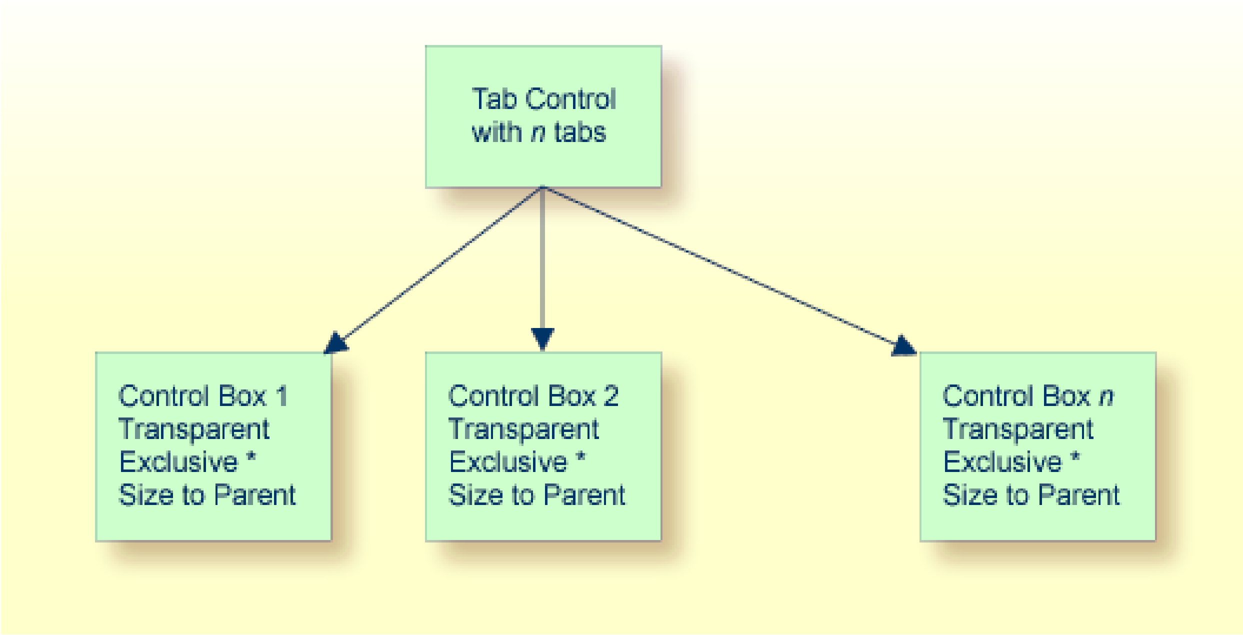
Note, however, that the dialog editor only automatically sets the child control's `OWNER` attribute if the tab control's "UI active (U)" `STYLE` flag is set, which is the default setting. Otherwise the child control's `OWNER` attribute is left unset (i.e., `NULL-HANDLE`). In the latter case, the child control is not automatically shown and hidden when switching between tabs. Note that the "UI active (U)" `STYLE` has no effect at run-time.

Use of Control Boxes as Tab Control Pages

As stated above, all child controls within a tab control have a tab control as their parent, regardless of the tab to which they belong. Whilst this is sufficient, it may be preferable to separate the controls on different tabs into separate sub-hierarchies.

The most convenient way of achieving this is to create a child control box for each tab to represent the tab „pages“. All other child controls are then created as child controls of the respective control box. Assuming that the tab control's "UI active (U)" STYLE flag is set, the control boxes will be automatically hidden and shown during tab switching, and thus t heir respective child controls with them (child controls are automatically hidden if any ancestor window is hidden). Otherwise, the program must do the page switching explicitly, as described in the next section.

The control's organization is shown in the following diagram:



As shown in the diagram, each child control box should be transparent, that the tab control's background texture (if any) shows through, and have the "size to parent (z)" STYLE, so that the control boxes automatically exactly fill the interior area of the tab control, both immediately and whenever the size of the tab control is changed. In addition, each control box should be „exclusive“ if the tab control is not UI active, such that only one child control box is visible at any time.

Switching Between Controls Belonging To Different Tabs

Anmerkung: This section only relates to tab controls that are not UI active. Otherwise, the control switching is done automatically by Natural.

In the dialog editor, this is performed automatically by the dialog editor when a control belonging to an exclusive control box (or the control box itself) is selected, which is not currently being displayed. The dialog editor makes the currently visible exclusive control box (if any) invisible (thus also hiding any controls placed within it) and makes the control box containing the selected control visible (thus also showing any controls placed within it). This process is independent of how the selection is made (for example, explicitly, from the selection box in the status bar, or implicitly, by simply tabbing through the controls).

Anmerkung: If the status bar is not shown, set the **Status Bar** under the **Dialog Editor** tab of the **Options** dialog opened via the **Tools > Options** command.

At run-time, the control boxes must, of course, be shown or hidden in response to the user selecting the corresponding tab. This can be achieved by querying the active tab in the tab control's `CHANGE` event and setting the `VISIBLE` attribute of the corresponding control box to `TRUE`. One way of making the tab/control box associations is to store the handle of the control box in the `CLIENT-HANDLE` attribute of the corresponding tab in the `AFTER-OPEN` event of the dialog.

For example:

```
/* Map control boxes to tabs:
#TAB-1.CLIENT-HANDLE := #CTLBOX-1
#TAB-2.CLIENT-HANDLE := #CTLBOX-2
..
#TAB-N.CLIENT-HANDLE := #CTLBOX-N
```

Then, assuming `#CONTROL` is defined as `HANDLE OF GUI`, the `CHANGE` event of the tab control (`#TABCTRL-1`) could look like this:

```
/* Get active tab
#CONTROL := #TABCTRL-1.SELECTED-SUCCESSOR
/* Switch to control box belonging to active tab:
#CONTROL := #CONTROL.CLIENT-HANDLE
IF #CONTROL <> NULL-HANDLE
  #CONTROL.VISIBLE := TRUE
END-IF
```

Mixing Tab-dependent and Tab-independent Controls

In some situations, it may be desirable to display controls that remain visible, irrespective of which tab is currently selected.

There are two ways of achieving this:

1. If control boxes are being used, the control boxes can be made smaller in order to cover only part of the tab control's interior area, leaving the remaining space available for controls that should be permanently displayed. The "size to parent (z)" `STYLE` must be switched off for the control boxes.
2. If control boxes are not being used and the tab control is UI active, permanently displayed controls may be created by ensuring that the "UI active (U)" `STYLE` for the tab control is temporarily switched off whilst creating the child control(s) that are to be permanently displayed.

If the tab control is not UI active, a two-layer control box hierarchy can be used, where the child control boxes described above are created as child controls of a transparent top-level control box, which in turn is created as a child of the tab control. The top-level control box (which does *not* have the "size to parent" flag set), can then be positioned and sized appropriately to define the replaceable region.

Anmerkung: This is very similar to the technique used for wizard dialogs. See the section *Working with Control Boxes* for more information

Keyboard Navigation

There are three methods of navigating between the tabs of a tab control via the keyboard, any combination of which can be applied simultaneously:

1. If the tab control is assigned the "browsable (z)" `STYLE` flag, the tab control is included in the tab sequence (i.e., can be navigated to via the TAB key). When the tab control receives the focus, navigation between the tabs is possible via the arrow keys. There is no „wrap-around“ between the first and last tabs in this case.
2. The tab captions (`STRING` attribute) may contain an ampersand (&), indicating that the following character is a mnemonic character. The tab is selected when the mnemonic character is pressed together with the ALT key. This allows for „direct“ keyboard navigation to the desired tab.
3. If the dialog has the "Property Sheet (p)" `STYLE` set, the keyboard shortcuts CTRL+TAB and CTRL+SHIFT+TAB may be used to navigate to the next and previous tab (respectively), with wrap-around.

Note that the focus does not have to be on the tab control or a dialog element within it in order for the last technique to work. Starting from the focus control, Natural examines each container (ancestor), until a container (if any) is found that contains one or more tab controls. If this container contains exactly one tab control, the keyboard shortcuts are then applied to this tab control. If it contains two or more tab controls, these shortcuts have no effect. If the dialog does not contain a tab control, the shortcuts perform their usual function, as if the "Property Sheet (p)" STYLE had not been set.

Note that this default usage of the CTRL+TAB and CTRL+SHIFT+TAB key combinations may be overridden by redefining them via the ACCELERATOR attribute.

Tab Switching Events

Whenever a tab switch is performed (either by the user or programmatically), the following sequence of events occurs:

1. The currently selected tab receives a LEAVE event, if not suppressed. This event is typically used for data validation and/or committing the data on the tab page.
2. The MODIFIABLE attribute of the tab control is then examined. If it is set to FALSE, the tab switch is not performed. The currently selected tab remains selected and no further events are raised. This can be useful if data validation performed during the LEAVE event found an error, which should be corrected by the user before continuing.
3. All direct child controls (if any) that have currently selected tab as their OWNER are automatically hidden.
4. The new tab is selected.
5. All direct child controls (if any) that have the newly-selected tab as their OWNER are automatically shown, if their VISIBLE attribute is set to TRUE.
6. The newly-selected tab receives an ENTER event. If not suppressed. This event is typically used for initializing controls on the new tab page.
7. The tab control receives a CHANGE event. This is convenient for tracking tab switches and responding to them without having to modify the event handlers for each tab.

Note that no initial ENTER event is raised for the selected tab when the control is created, and that the tab control does not receive an initial CHANGE event either. Furthermore, when the dialog containing the tab control is closed, the currently-selected tab does not receive a LEAVE event.

107 Working with Tree View Controls

- Introduction 794
- Setting Item Images 794
- Item Selection 795
- Item Activation 795
- Item Data 796
- Sorting 797
- Label Editing 797
- Multiple Context Menus 798
- Dynamic Item Creation 799
- Drag and Drop 800

Introduction

A tree view control is used to display data in hierarchical form. Each node in the hierarchy is represented internally as a tree view item. The following shows a simple tree view (with optional check boxes) displaying a 3-level hierarchy containing seven tree view items:

The tree view control shown above is specified with the "+/- buttons (b)", "Lines (l)", "Lines at root (r)" and "Check boxes (c)" STYLE flags.

The height of each item and the indentation between levels can be set via the ITEM-H and SPACING attributes, respectively. If either of these are zero, the default setting is used.

Setting Item Images

Images for the tree view items may be defined by creating and associating an image list control with the tree view control, then (for each item) selecting the required image from the image list via its index and/or image handle, as described in the section ***Working with Image List Controls***.

Please note that it is not necessary to set the image list control's "Large images (L)" style, unless you are additionally using the same image list for a list view control, because the tree view control only uses small images.

Item Selection

Unlike the list view control, the tree view control only supports one selected item. The current selection (if any) may be retrieved by querying the tree view control's (read-only) SELECTED-SUCCESSOR attribute.

In addition to being selected by the user, items may be selected or deselected programmatically by setting or clearing their SELECTED attribute.

When an item is selected, a CLICK event is raised for the list view control (if not suppressed), with the handle of the corresponding item being set in the control's ITEM attribute.

Item Activation

When a user double-clicks on an item, an ACTIVATE event is raised (unless suppressed) for the tree view control. The application, if it decides to handle this event, normally performs a user-defined default action on the selected item, the handle to which may be obtained via either the ITEM or SELECTED-SUCCESSOR attributes of the tree view control. Note that there may be other, non-default, actions applicable to the selected item, but these are typically accessed via other mechanisms. For example, they may be listed (typically along with the default action) in a context menu displayed by the application.

If the "Dbl. click expand (d)" STYLE flag is set, double clicking a tree view item that has child items expands the item in addition to activating it.

The ACTIVATE event can also be triggered via the keyboard. This can be done in either of two ways:

1. By pressing the key or key combination defined for the tree view control's ACCELERATOR attribute. The control need not currently have the focus.
2. By pressing the ENTER key, if the tree view control currently has the focus. This method only works if the dialog neither contains a default pushbutton, nor a pushbutton with the "OK Button (O)" STYLE flag set.

In either case, no ACTIVATE event is raised if no item is currently selected.

Item Data

In order to be able to support sorting correctly (see next section), a tree view item's data does not need to be alphanumeric, as it is per default, but can be one of any of the pre-defined types supported by the column's FORMAT attribute. In addition, an edit mask can be applied to the item by setting its EDIT-MASK attribute. The item's label, as seen by the user for the item, is the alphanumeric representation of the item's internal data, using the associated edit mask (if any). The conversion between the item's internal data and the displayed data is compatible with the MOVE EDITED statement if an edit mask is supplied. Otherwise, the conversion between the internal and displayed data, and vice-versa, is compatible with the conversion involved in copying the data to and from the Natural stack (see the STACK TOP DATA and INPUT statements). For example, numeric values are displayed using the current decimal character (as defined by the DC parameter) if necessary, with a leading minus character if negative, date values are displayed in the format defined by the DTFORM parameter setting, logical values are displayed as an "X" if true and as a blank if false, and so on.

An example of use of a non-alpha tree view item is shown below:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #TVITEM-DATE
   TYPE = TREEVIEWITEM
   PARENT = #TV-1
   STRING = '+123.456'
   FORMAT = FT-DECIMAL
   EDIT-MASK = '+ZZZ,ZZ9.999'
END-PARAMETERS GIVING *ERROR
```

In this case, the specified item label (STRING attribute value) must of course be a valid number of a form compatible with the specified edit mask (+ZZZ,ZZ9.999). Internally, the label is converted to the data type and length corresponding to the specified format, FT-DECIMAL (= P10.5).

Note that it is not possible to access the underlying data for a tree view item directly. The item's data can only be set or retrieved via the item's label.

Sorting

Tree view items can either be inserted in sorted sequence or explicitly sorted after insertion, by calling the `SORT-ITEMS` action. Items are inserted in ascending alphabetical sequence if either the tree view control or the tree view item being inserted have their `SORTED` attribute set to `TRUE`. In the latter case, the items are optionally sorted in ascending or descending sequence according to their internal data (see last section). See the documentation for the `SORT-ITEMS` action for more information. Note that it is the responsibility of the programmer to ensure that the underlying item formats of the items concerned (as defined by the `FORMAT` attribute) are of comparable types. For example, it is possible to mix integers and floating point values, but not integers and dates.

Label Editing

The process of label editing for tree view controls is the same as for list view controls. Therefore, for more information on this subject, please refer to the section ***Label Editing in Tree View and List View Controls***.

Note that even if the `SORTED` attribute is set, the items are not automatically re-sequenced after a label editing operation has been completed. If this is required, this can be done as shown in the example below. Firstly, we define some variables for later use:

```
01 #CONTROL HANDLE OF GUI 01 #ITEM HANDLE OF TREEVIEWITEM
01 #SORTITEM HANDLE OF TREEVIEWITEM
```

In addition, it is assumed that the tree view control is named `#TV-1`.

In the tree view control's `AFTER-EDIT` event, we cannot do the re-sequencing asynchronously, as this would interfere with the (as yet incomplete) editing process. Instead, we simply set the `#SORTITEM` variable to indicate that the re-sequencing should occur at a later time, after the editing process has been completed. This only needs to be done if the tree view items are sorted:

```
#CONTROL := *CONTROL #ITEM := #CONTROL.ITEM IF #CONTROL.SORTED OR #ITEM.SORTED
#SORTITEM := #ITEM ELSE #SORTITEM := NULL-HANDLE END-IF
```

Note that this examples assumes the use of the default (ascending alphabetical) sorting provided by the tree view control iteself.

The actual work of re-sequencing the items is done asynchronously in the dialog's `IDLE` event:

```
IF #SORTITEM <> NULL-HANDLE PROCESS GUI ACTION SORT-ITEMS WITH #SORTITEM.PARENT
GIVING *ERROR RESET #SORTITEM END-IF
```

Multiple Context Menus

If you wish to support just a single context menu for the control, you can simply set the control's `CONTEXT-MENU` attribute to the handle of the context menu you wish to display, and leave it set to this value. However, it is often required to be able to display more than one context menu for list view controls, whereby this approach is too inflexible.

To address the above problem, the `CONTEXT-MENU` event was introduced (not to be confused with the attribute of the same name as mentioned above!). This event (if not suppressed) is raised for the target control immediately before its `CONTEXT-MENU` attribute is evaluated, allowing the application to dynamically set this attribute to the handle of the appropriate context menu first.

As an example, assume that we have defined two context menus in the dialog editor: one containing item-related commands, `#CTXMENU-ITEMS`, and one containing generic commands (e.g., for switching the view mode), `#CTXMENU-DEFAULT`. In this case, the following `CONTEXT-MENU` event could be used:

```
#CONTROL := *CONTROL IF #CONTROL.SELECTED-SUCCESSOR
<> NULL-HANDLE #CONTROL.CONTEXT-MENU := #CTXMENU-ITEMS ELSE #CONTROL.CONTEXT-MENU
:= #CTXMENU-DEFAULT END-IF
```

where the following local data definition is assumed:

```
01 #CONTROL HANDLE OF GUI
```

In this example, the context menu `#CTXMENU-ITEMS` will be displayed if the user right clicks at a position occupied by an item, or `#CTXMENU-DEFAULT` otherwise.

Of course, this technique can be refined further to display context menus specific to the type(s) of the selected item(s).

Dynamic Item Creation

When a tree view item is expanded or collapsed, an EXPAND event or COLLAPSE event (respectively) is raised for the control, if the event is not suppressed. Amongst other things, these events allow tree-view items to be dynamically created and deleted on demand.

For example, the following code demonstrates the dynamic creation of three items in response to the EXPAND event. It assumes that a dummy placeholder item with an empty STRING attribute value is already in place at the position where the items should be inserted. The placeholder can either have been statically defined in the dialog editor, or dynamically-defined during the initial tree view item creation. The purpose of the placeholder is to ensure that a "+" button (if button display enabled via the "+/- buttons (b)" STYLE) appears next to the parent node.

The following EXPAND event code assumes that the variable #TGT-ITEM contains the handle of the tree view item under which the dynamic tree view items are to be created:

```
#CONTROL := *CONTROL
#ITEM := #CONTROL.ITEM
IF #ITEM = #TGT-ITEM
  #TVITEM-DYN := #ITEM.FIRST-CHILD
  IF #TVITEM-DYN <> NULL-HANDLE AND
     #TVITEM-DYN.STRING = ' '
    PROCESS GUI ACTION DELETE WITH #TVITEM-DYN GIVING *ERROR
    FOR #I 1 3
      COMPRESS 'Dynamic Item' #I INTO #A
      PROCESS GUI ACTION ADD WITH PARAMETERS
        HANDLE-VARIABLE = #TVITEM-DYN
        TYPE = TREEVIEWITEM
        PARENT = #ITEM
        STRING = #A
      END-PARAMETERS GIVING *ERROR
    END-FOR
  END-IF
END-IF
```

where the following local data definitions are assumed:

```
01 #CONTROL HANDLE OF GUI
01 #ITEM HANDLE OF TREEVIEWITEM
01 #TVITEM-DYN HANDLE OF TREEVIEWITEM
01 #TGT-ITEM HANDLE OF TREEVIEWITEM
01 #I (I4)
01 #A (A) DYNAMIC
```

The above code first looks to see whether the item being expanded is the target item. If so, it queries the `STRING` attribute of the first child, to find out whether a placeholder is present. If this is the case, the placeholder is deleted, and three dynamic tree view items are created. The `STRING` attribute value for the inserted items is specified on creation, rather than modified afterwards, to ensure that the code also works correctly for `SORTED` tree views.

To save resources, the dynamically-created items could optionally be deleted again in the `COLLAPSE` event handler, being replaced by a placeholder item:

```
#CONTROL := *CONTROL
#ITEM := #CONTROL.ITEM
IF #ITEM = #TGT-ITEM
  PROCESS GUI ACTION DELETE-CHILDREN WITH #ITEM GIVING *ERROR
  PROCESS GUI ACTION ADD WITH PARAMETERS /* placeholder
    TYPE = TREEVIEWITEM
    PARENT = #ITEM
  END-PARAMETERS GIVING *ERROR
END-IF
```

Drag and Drop

The basic technique for providing drag and drop support is described in the section ***Using the Clipboard and Drag and Drop***.

Note that it is the responsibility of the programmer to (if required) highlight the item (if any) under the mouse cursor by setting its `SELECTED` attribute, and to restore the original selection (if any) afterwards.

The following example provides some code for demonstrating a tree view control acting as a drop target, in order to support dragging and dropping text from another application (e.g. WordPad) onto a tree view item in order to change its label.

The first step is to ensure that the drop mode is set correctly for the tree view control. In the **Tree View Control Attributes** window in the dialog editor, set the **Drop mode** selection box to

Copy+Move. This causes the control's DROP-MODE attribute to be set to DM-COPYMOVE in the generated source code for the dialog.

Next, the required local variables that are going to be used below must be defined:

```
01 #CONTROL HANDLE OF GUI
01 #DROP-ITEM HANDLE OF GUI
01 #ITEM HANDLE OF GUI
01 #SELECTED HANDLE OF GUI
01 #AVAIL (L)
01 #X (I4)
01 #Y (I4)
```

Having done this, we can write the necessary event handlers. The logical place to start is with the DRAG-ENTER event:

```
#CONTROL := *CONTROL
#CONTROL.CLIENT-HANDLE := #CONTROL.SELECTED-SUCCESSOR
PROCESS GUI ACTION INQ-FORMAT-AVAILABLE WITH CF-TEXT #AVAIL GIVING *ERROR
IF #AVAIL
  #CONTROL.SUPPRESS-DRAG-DROP-EVENT := NOT-SUPPRESSED
ELSE
  #CONTROL.SUPPRESS-DRAG-DROP-EVENT := SUPPRESSED
END-IF
```

The above code first saves the handle of the currently selected item in the control's CLIENT-HANDLE attribute for the purposes of restoring the selection to this item later. The event handler then checks whether text data is available on the drag-drop clipboard. If it is, the DRAG-DROP event is unsuppressed in order to allow a drop. Otherwise, we suppress this event in order to prohibit a drop such that the „no drop“ drag cursor appears.

To provide drop emphasis during the dragging of external data across the tree view control, a DRAG-OVER event handler is supplied:

```
#CONTROL := *CONTROL
IF #CONTROL.SUPPRESS-DRAG-DROP-EVENT = NOT-SUPPRESSED
  PROCESS GUI ACTION INQ-DRAG-DROP WITH
    3X #X #Y GIVING *ERROR
  PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
    #CONTROL #X #Y #ITEM GIVING *ERROR
  IF #ITEM <> #DROP-ITEM
    #DROP-ITEM := #ITEM
    IF #DROP-ITEM <> NULL-HANDLE
      #DROP-ITEM.SELECTED := TRUE
    END-IF
```

```
  END-IF
END-IF
```

The above code performs the following:

1. If a drop was disallowed in the `DRAG-ENTER` event, the event is ignored, as no drag emphasis is required in this case.
2. Otherwise, the `INQ-DRAG-DROP` action is called to determine the current drop position.
3. The `INQ-ITEM-BY-POSITION` action is used to find the tree view item (if any) at the current drop position. This item is tracked in `#DROP-ITEM`.
4. The tree view item (if any) under the cursor is selected to provide the drop emphasis by setting its `SELECTED` attribute to `TRUE`.

To perform the actual drop, a `DRAG-DROP` event handler is supplied:

```
IF #DROP-ITEM <> NULL-HANDLE
  PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH CF-TEXT #DROP-ITEM.STRING
    GIVING *ERROR
END-IF
#CONTROL := CONTROL  /* "parameter" for subroutine below
PERFORM RESET-SELECTION
```

If there is a tree view item representing the drop target, the above code retrieves the text from the drag-drop clipboard directly into the item's caption. Afterwards, the original selection state of the tree view control is restored. The `RESET-SELECTION` subroutine used for this purpose can be written as follows:

```
#ITEM := #CONTROL.CLIENT-HANDLE
IF #ITEM <> NULL-HANDLE
  /* Restore original selection:
  #ITEM.SELECTED := TRUE
ELSE
  /* Nothing was originally selected,
  /* so clear any existing selection:
  #ITEM := #CONTROL.SELECTED-SUCCESSOR
  #ITEM.SELECTED := FALSE
END-IF
RESET #DROP-ITEM
```

To satisfy the logic provided for the other event handlers, `#DROP-ITEM` is also reset, indicating the absence of a drop item.

Lastly, in the case that the user cancels the drag operation, or exits the bounds of the tree view control without having performed a drop, a DRAG-LEAVE event handler is supplied:

```
#CONTROL := CONTROL  /* "parameter" for subroutine below
PERFORM RESET-SELECTION
```

The above code simply invokes the inline subroutine listed above in order to clear the drop emphasis (if any), and to restore the original selection state.

108 Working with Dynamic Information Line and Status Bar

When you are working with both a dynamic information line (DIL) and a status bar, the combination of the HAS-DIL and HAS-STATUS-BAR attributes determines whether and when DIL-TEXT and STATUS-TEXT values will be displayed:

HAS-DIL	HAS-STATUS-BAR	DIL-TEXT	STATUS-TEXT
TRUE	TRUE	displayed	displayed
TRUE	FALSE	-	-
FALSE	TRUE	-	displayed
FALSE	FALSE	-	-

If HAS-DIL and HAS-STATUS-BAR are TRUE, the DIL-TEXT will overlap the STATUS-TEXT value and vice versa, depending on which was modified last.

109 Adding a Maximize/Minimize/System Button

To add a Maximize/Minimize/System button to your dialog

- Open the **Dialog Attributes** window. Check the **System Button** or **Maximizable** or **Minimizable** entry.

When the **System Button** entry is checked, the dialog's standard control menu is available. This includes the control menu box (to close the dialog), the title bar, and the Maximize and Minimize buttons.

110 Defining Color

You can define colors for dialogs and dialog elements. These can be foreground colors and background colors. To do this, you use the following attributes:

- BACKGROUND-COLOUR-NAME
- BACKGROUND-COLOUR-VALUE
- FOREGROUND-COLOUR-NAME
- FOREGROUND-COLOUR-VALUE

You can assign only standard colors to the attributes ending with NAME. The attributes ending on VALUE, however, can be assigned customized colors following the RGB model.

You can set colors:

- in an attributes window, or
- in event-handler code.

You can directly assign a value to the attributes ending with NAME. If you want to assign a value to an attribute ending with VALUE, you must set the NAME attribute to the value CUSTOM. If you do not set the NAME attribute to the value CUSTOM, the VALUE attribute is ignored.

Examples:

```
#DIA.BACKGROUND-COLOUR-NAME:= MAGENTA     /* Assign a value to a NAME
                                          /* attribute
#DIA.BACKGROUND-COLOUR-NAME:= CUSTOM      /* Set NAME to CUSTOM
#DIA.BACKGROUND-COLOUR-VALUE:= H'FF0000'  /* Then assign Red, Green, and
                                          /* Blue values to the VALUE
                                          /* attribute (hexadecimal)
```

Anmerkung: You can not use all customized colors in all parts of the user interface. Colors in text, for example, must always be monochrome.

When setting a color in an attributes window, you have three possibilities:

- Use the attribute ending with `NAME` and leave the value at `DEFAULT`. You can also do this in code. Your color will then be determined by your color settings in the windowing system.
- Use the attribute ending with `NAME` by pulling down the list box and choose one of the predefined colors.
- Define your own color by using the attribute ending with `VALUE`.

To define a color

1. Choose the **Custom** push-button control right of the **Background color** entry. A dialog box appears.
2. Select one of the predefined colors or choose the **Define Custom Colors** push-button control. To set the red, green, and blue values, use the cursor to select the desired color or enter a value from 1 to 253 in the red, green, and blue value display fields.
3. Choose the **Add to Custom Color** push button control. To save the newly defined color, choose the **OK** button in the dialog box. The newly defined color is now selected by default.
4. To set it, close the attributes window.

111 Adding Text in a Certain Font

To choose a specific font for the text assigned to a dialog element (for example, the caption on a push button control)

1. Use the dialog element's attributes window.
2. Choose the **...** push button control to the right of the **Font** entry. A dialog box opens.
3. From the list of available fonts, select a font type, for example **Times New Roman**.
4. From the list of styles available for the font type, select a font style, for example **italics**.
5. From the list of sizes available for the font type and style, select a font size, for example **10**. A sample of your selected font will be displayed.
6. To set it: Close the attributes window.

Anmerkung: When adding centered or right-aligned text in a dialog element, the following minimum heights of the dialog element apply (`RECTANGLE-H` attribute): 4-point font - height of 8; 8-point - 22; 12-point - 24.

Additionally, the dialog editor allows selecting a font for the whole dialog in the dialog attributes window. This font is defined in the `FONT-STRING` attribute and is valid for the dialog and each of its children. A major advantage of selecting a font for the whole dialog is that if the chosen font is too large or too small for the dialog layout, you change the `FONT-STRING` attribute once instead of going through all children of the dialog.

Initially, the `FONT-STRING` attribute must be set as a parameter while the dialog is being created with `PROCESS GUI` action `ADD`. If a dialog element inside the dialog contains text with no particular font assigned to it, this text will be displayed in the font specified by `FONT-STRING`.

112 Adding Online Help

From an application written with the dialog editor, you can invoke help for a specified help topic ID. Please bear in mind that you will have to create parts of the help associated with these help topic IDs outside the Natural development environment. You will also have to compile the help with the platform-specific help compiler.

To keep an overview of all the different help sections in an application, Natural provides you with the help organizer. With this organizer,

- you assign a help ID (`HELP-ID` attribute value) to a specific dialog element;
- you write the help text for the associated help topic; this text is converted to a *.rtf* file to be processed by the help compiler;
- you optionally define the help topic's keywords;
- you optionally assign a help compiler macro to the help topic;
- and optionally you add a comment for your internal documentation purposes.

To create a help topic

1. Invoke the help organizer's main dialog.
2. Select a particular dialog element.
3. Generate a new help topic ID.
4. Return to the help organizer main dialog.
5. Assign the generated help topic ID.
6. Enter the external definitions for the help topic ID, such as the help topic text and the topic name.
7. Return to the help organizer main dialog.

8 Go to the topic list and see whether this new help topic fits your general organization of the help file to be created.

9 Return to the help organizer main dialog.

10 Save everything.

A dialog or dialog element can also be assigned a `HELP-ID` number independently of the help organizer.

To do so

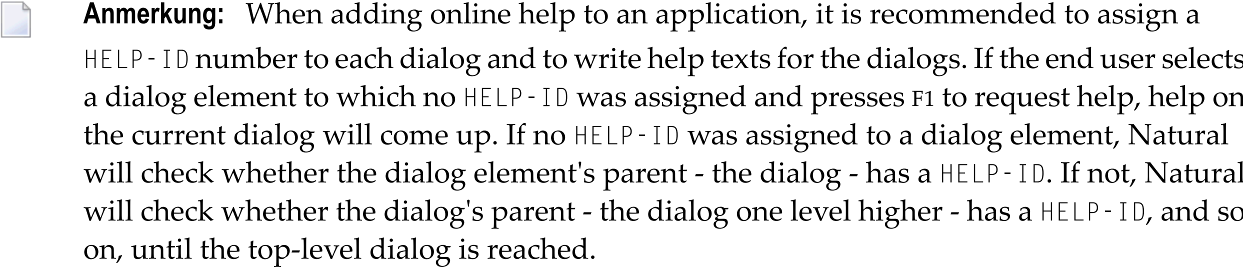
- Open the corresponding attributes window. Enter a numeric value in the **Help ID** entry.

You must use the help topic's *.h* file to map the numerical ID that you enter here to the corresponding help topic ID (created by a markup in the *.hlp* file).

Natural expects the help file to be located in the resource (RES) subdirectory of the current library or one of the STEPLIBs, or in the directory referred to by the environment variable `NATGUI_BMP`. By default, Natural searches for a help file with the same name as the current library, but you can explicitely set the name of the help file via the `HELP-FILENAME` attribute. If no file extension is specified, Natural searches for a compiled HTML help file with the extension ".chm" first, then (if not found) for a WinHelp help file with the extension ".hlp". Thus, if no file extension is specified, it is possible to upgrade from using WinHelp to using HTML help without changing the Natural program. Note, however, that the Help Organizer only supports WinHelp. If you wish to create HTML help content, you must use an external help authoring tool to do so.

Whenever an end user presses F1 in an active dialog, Natural first queries for a file with the value of the `HELP-FILENAME` attribute plus the extension ".h*nn*" where *nn* is the Natural language code. If it does not find such a file, it queries for a file with the value of `HELP-FILENAME` plus the extension ".hlp"

Whenever the dialog element has the focus and the end user presses F1, Natural jumps to this help ID.



Anmerkung: When adding online help to an application, it is recommended to assign a `HELP-ID` number to each dialog and to write help texts for the dialogs. If the end user selects a dialog element to which no `HELP-ID` was assigned and presses F1 to request help, help on the current dialog will come up. If no `HELP-ID` was assigned to a dialog element, Natural will check whether the dialog element's parent - the dialog - has a `HELP-ID`. If not, Natural will check whether the dialog's parent - the dialog one level higher - has a `HELP-ID`, and so on, until the top-level dialog is reached.

To build a help file

1 Go to your command promt.

2 Change to the directory referred to by the environment variable `$NATGUI_BMP`.

3 Issue the command `HCRTF -X` *`helpfilename`*.

Anmerkung: This assures that the directory containing *HCRTF.EXE* is specified in the `PATH` environment variable.

To test a help file

1 Invoke a dialog in your application.

2 Press F1.

The help topic for the dialog should appear.

Alternatively, the help file can be conveniently built and tested interactively by opening the *.hpj* file in the Help Compiler Workshop (*HCW.EXE*).

To display help in a popup window

1 Check the **Popup Help** option in the dialog attributes window.

2 Run the dialog.

3 Press F1 with the focus on a control which has a help ID associated with it.

The help topic associated with the focus control should appear in a popup window.

113 Defining Mnemonic and Accelerator Keys

- Introduction 818
- Defining a Mnemonic Key 818
- Defining an Accelerator Key 819
- Displaying Accelerator Keys in Menus 819

This document covers the following topics:

Introduction

There are two ways of providing keyboard commands:

- A mnemonic key is determined by an underlined character in a visible dialog element, for example a menu item. The end user can select the menu item by pressing „ALT+mnemonic key“, for example ALT+A.
- An accelerator key is defined in the `ACCELERATOR` attribute. By pressing this key, the end user causes a double-click or click event for the dialog element regardless of whether the dialog element is visible or not, as long as the dialog element is enabled.

Defining a Mnemonic Key

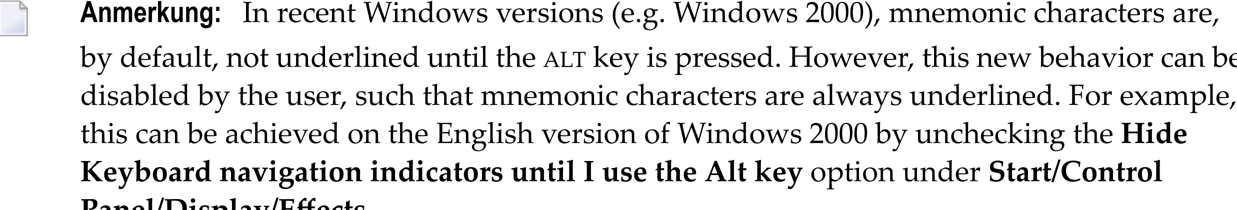
You define a mnemonic key in the dialog element's `STRING` attribute by specifying "&" before the desired character. At runtime, the character will be underlined. Example: the `STRING` attribute value "E&xplanation" will be displayed as follows at runtime:

Explanation

If you define a mnemonic key with a text constant control or a group frame control, and the end user presses the mnemonic key at runtime, the next dialog element in the control sequence will get the focus. For example, if the next dialog element after a text constant control is an input-field control, the text constant control's mnemonic key sets the focus to the input field control. Whenever you disable such an input field control at runtime, you should also disable the corresponding text constant control.

You can define mnemonic keys in the `STRING` attribute of the following types of dialog elements: group frame control, menu item, push button control, radio button control, text constant control, toggle button control, tool bar item.

You can still display an "&" in your runtime `STRING` by specifying "&&". Example: "A&&B" will be displayed as "A&B".



Anmerkung: In recent Windows versions (e.g. Windows 2000), mnemonic characters are, by default, not underlined until the ALT key is pressed. However, this new behavior can be disabled by the user, such that mnemonic characters are always underlined. For example, this can be achieved on the English version of Windows 2000 by unchecking the **Hide Keyboard navigation indicators until I use the Alt key** option under **Start/Control Panel/Display/Effects**.

Defining an Accelerator Key

You define an accelerator key by setting the ACCELERATOR attribute to a key or a key combination for the dialog element, for example to F6 or CTRL+1. If the end user presses the accelerator key, the double-click event occurs for the dialog element, or if no double-click event is available, the click event occurs. The accelerator key does not work if the corresponding event is suppressed, or if the dialog element is disabled.

Standard system accelerators such as ALT+ESC, CTRL+ESC, ALT+TAB and CTRL+ALT+DEL can be defined as accelerators, but do not cause the dialog element's click or double-click event to be triggered. Instead, they cause the associated system functionality to be invoked. The same applies to standard MDI accelerators (such as CTRL+F4 and CTRL+F6) if used within MDI applications and to any accelerators belonging to in-place activated servers (e.g. ActiveX controls which currently have the focus).

Note that user-defined accelerator keys overwrite identical user-defined shortcut keys associated with desktop items.

If the same accelerator key is associated with more than one dialog element, the dialog element whose click or double-click event is triggered is not defined.

A dialog element which references another via its SAME-AS attribute inherits the accelerator of the referenced object. For example, if a menu item references a signal, and the signal's accelerator is CTRL+ALT+X, then querying the menu item's ACCELERATOR attribute will also return CTRL+ALT+X. However, the accelerator, if pressed, will only trigger a click event for the referenced dialog element (i.e., the signal in this example).

Accelerators of the form ALT+X, where "X" is one of the alphabetic characters, should be avoided, because they are reserved for use as keyboard mnemonics.

Displaying Accelerator Keys in Menus

In order to show accelerators for menu items, the menu text needs to first be appended with a tab (h'09') character and then appended with the text for the accelerator. This cannot be done statically in the dialog editor's menu editor, because there is no way to enter a tab character into the string definition. However, the accelerators may be appended dynamically using a generic piece of code which iterates round all menu items for a dialog. This is illustrated by the following external subroutine, which can conveniently be called from within a dialog's AFTER-OPEN event.

```
DEFINE DATA
PARAMETER
  1 #DLG$WINDOW  HANDLE OF WINDOW
LOCAL
  1 #CONTROL     HANDLE OF GUI
  1 #COMMAND     HANDLE OF GUI
LOCAL USING NGULKEY1
END-DEFINE
*
DEFINE SUBROUTINE APPEND-ACCELERATORS
#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  IF #CONTROL.TYPE = SUBMENU OR #CONTROL.TYPE = CONTEXTMENU
    #COMMAND := #CONTROL.FIRST-CHILD
    REPEAT UNTIL #COMMAND = NULL-HANDLE
      IF #COMMAND.ACCELERATOR <> ' '
       COMPRESS #COMMAND.STRING H'09' #COMMAND.ACCELERATOR INTO
         #COMMAND.STRING LEAVING NO SPACE
      END-IF
      #COMMAND := #COMMAND.SUCCESSOR
    END-REPEAT
  END-IF
 #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
END-SUBROUTINE
END
```

This dynamic technique has the advantage that the accelerator does not, in effect, have to be defined twice (i.e., for the `ACCELERATOR` and `STRING` attributes of the menu item).

Note that if the target language is not English, the `ACCELERATOR` attribute value will probably have to be translated before being appended to the menu item string.

114 Dynamic Data Exchange - DDE

- Concepts 822
- Developing a DDE Server Application 823
- Developing a DDE Client Application 824
- Return Codes 825

Concepts

DDE is a protocol defined by Microsoft Corp. to enable different applications to exchange data. This means that, for example, an application written in Natural may exchange data with a spreadsheet, because they are both able to process the DDE protocol. An application that processes the DDE protocol communicates with another DDE application via standardized messages. One of the applications is defined as the client, the other as the server. Client and server are holding a DDE conversation.

Anmerkung: For an overview of DDE concepts and terminology, see your Microsoft Windows documentation.

Data in a DDE conversation is identified by a three-level hierarchy:

- service,
- topic,
- item.

A DDE conversation is established whenever a client requests a *service* from a DDE server. A DDE server offers one or more *services* to all active applications.

For each service, a DDE server may offer any number of *topics*. The DDE client then requests a conversation on a *topic* of a *service*.

In a conversation on a *topic* of a *service*, the DDE client and the DDE server uniquely identify data to be exchanged by an *item* name.

A DDE server may support a number of services, which in turn may consist of a number of topics, which themselves may contain a number of items.

With Natural, you can develop both DDE client applications as well as DDE server applications. You may, for example, write a Natural DDE client application that requests data from a spreadsheet acting as a DDE server, or you may write a Natural DDE server application that supplies a word processor (DDE client) with data.

To develop DDE client and DDE server applications, the following functionality is provided:

- A number of NGU-prefixed subprograms in library SYSTEM; these send messages and data as defined in the parameter data area NGULDDE1
- a parameter data area (NGULDDE1) which describes the parameters used by the subprograms in a DDE conversation (the DDE-VIEW);
- a DDE-Client event and a DDE-Server event which handle DDE messages.

You develop a DDE server application by reacting to the DDE-Server event and by using the NGU-SERVER-prefixed subprograms from library SYSTEM to register services and topics and to send messages and data to the DDE client application.

You develop a DDE client application by reacting to the DDE-Client event and by using the NGU-CLIENT-prefixed subprograms from library SYSTEM to initiate conversations and send requests and other DDE commands to DDE server applications.

You always have to include the parameter data area NGULDDE1 and the local data area NGULFCT1 in your client or server dialog. (You need NGULFCT1 in order to use the NGU-prefixed subprograms in library SYSTEM).

Developing a DDE Server Application

The following topics are covered below:

- Registering/Unregistering Services and Topics
- Getting Data From The Client
- Sending Data To The Client
- Terminating DDE Server Operation

Registering/Unregistering Services and Topics

Before a DDE server application can be addressed by a DDE client application, it must register its service names and all supported topics for the services. You use subprogram NGU-SERVER-REGISTER to do this for each service/topic the DDE server supports. Registering will usually be handled in the „after open“ event of the base dialog.

When registering a service/topic for the first time, you will need to supply Natural with the dialog-ID of the dialog that will function as the server and that will therefore receive all DDE messages from clients. This is done by setting the `DDE-VIEW.CONV-ID` to the respective dialog-ID and also by setting `DDE-VIEW.MESSAGE` to the string "DLGID".

Note that at a later time you are able to add more topics to a service or even entirely new services. You can also make a topic unavailable by using subprogram NGU-SERVER-UNREGISTER.

Getting Data From The Client

After successful registration, it is possible that the DDE server application receives DDE messages from a DDE client application which is establishing a conversation on a registered topic of a service.

Such messages for a DDE server are received in the DDE-Server event of the dialog. At the beginning of the event-handler section, it is necessary to fill the DDE-VIEW with the client's message data. This is done by using subprogram NGU-SERVER-GET-DATA. After reading the data, it will be necessary to act based on the client message received. The possible messages and their meaning are explained in the description of subprogram NGU-SERVER-GET-DATA.

Sending Data To The Client

In many cases, the client message ultimately requires the server to send data to the client. This is achieved by using the subprogram NGU-SERVER-DATA.

Terminating DDE Server Operation

Whenever DDE server operation is supposed to terminate, you use the subprogram NGU-SERVER-STOP. It unregisters all services and terminates all active conversations. You terminate the server application with the `CLOSE DIALOG` statement.

Developing a DDE Client Application

The following topics are covered below:

- Connecting With The DDE Server Application
- Using The Services of a DDE Server Application
- Receiving Data From The DDE Server Application
- Disconnecting From The DDE Server Application
- Terminating DDE Client Operation

Connecting With The DDE Server Application

In order to establish a conversation with a DDE server application, a DDE client application must call the subprogram NGU-CLIENT-CONNECT with the service and topic name of the server it wants to connect. In order to receive the appropriate DDE events from a server, it is necessary to set the `DDE-VIEW.CONV-ID` to the client's dialog-ID and also to set `DDE-VIEW.MESSAGE` to the string "DLGID". The call will return a unique conversation ID in `DDE-VIEW.CONV-ID`. This value must be set appropriately in all further communication with the server.

Using The Services of a DDE Server Application

The client has several options to use the services of a server once a conversation has been established. It can

- request data on a specific item (using NGU-CLIENT-REQUEST),
- send data to the server (using NGU-CLIENT-POKE),
- ask the server to execute a command (using NGU-CLIENT-EXECUTE), or
- establish a warm or hot link to the server (using NGU-CLIENT-ADVISE-HOT, NGU-CLIENT-ADVISE-WARM and NGU-CLIENT-ADVISE-TERM).

Receiving Data From The DDE Server Application

The DDE client will receive data or other messages from the DDE server via the client dialog's DDE-Client event. Whenever a server has sent a message, this event occurs. The message contents must first be retrieved using NGU-CLIENT-GET-DATA. This will fill the DDE-VIEW structure appropriately. The client must then determine which message (DDE-VIEW.MESSAGE) has arrived and react appropriately. The possible messages are listed in the description of subprogram NGU-CLIENT-GET-DATA.

Disconnecting From The DDE Server Application

Whenever the client determines that the conversation is no longer needed, a call to NGU-CLIENT-DISCONNECT must be issued to inform the server that the conversation is to be terminated.

Terminating DDE Client Operation

Whenever the client application terminates or wants to stop using DDE, it needs to call NGU-CLIENT-STOP. This informs Natural to close all active conversations of the client and shut down DDE operation for the application.

Return Codes

Possible return codes are described in this section.

Anmerkung: Each error-code description is not necessarily comprehensive. In these cases, the description is marked with an asterisk (*).

Code	Meaning
-1	You have specified an incorrect command or command parameter. Ensure that your DDE data area is of the correct type and that the command is correct.
0	The function was processed correctly.
1	This value is returned when an application has attempted to initialize with the DDEML library more than once. Check the logic of your program. Also ensure that the DDEML was exited correctly during the last run of the program.
2	This value may be returned from the server-initialize function if you have run the program before and not exited the DDEML correctly. It is also returned by a call-back function, whenever the requested service failed.
	An error occurred in the underlying layer. *
3	The conversation ID referenced does not represent an active conversation. Check if you have specified a correct service name.
4	The application could not initialize with the DDE library as the maximum number of instances are connected.
5	The DDEML communication has not been initialized. You must initialize with the DDEML before any DDE activity can take place.
6	Memory allocation problems encountered. This error might occur if the queue of messages for either part in the conversation becomes too long. *
7	A service, topic or item name was longer than 255 characters. Check if your fields are correctly specified for DDE-VIEW and make sure that you are not attempting to place a string longer than 255 characters in any one of the above variables.
8	An error occurred in the DDE library. Contact Software AG Support. *
9	Parameters passed to this function were illegal. This can be returned by any function call. Check your parameters.
10	„Server Type Link“ is supported but no call-back function for UNLINK is passed to the function `PIDsRegisterTopic`.*
11	An attempt was made to remove a topic for which at least one conversation is still active. This includes trying to unregister a topic for which a conversation still exists.
12	The service/topic referenced has not been registered with the function `PIDsRegisterTopic`.
13	No links were active for the DDE-VIEW.SERVICE when the NGU-Server-Data subprogram was used. Check your service name and use the DDE-SPY in the SDK Tool Kit to see what services are available.
14	The requested type of link is invalid.
15	The transaction ID is corrupted. Check the value of your transaction ID in your DDE view.
16	The client application requested a conversation and prior to that, no function was specified to send the data for the links.
17	An asynchronous transaction was requested, but the client application did not specify a function to send details of the completed transaction. Such a function must be specified when the conversation is initialized.

Code	Meaning
18	A synchronous transaction timeout expired. The amount of time taken for your transaction to complete was longer than the TIMEOUT value in your DDE-VIEW structure. Increase the TIMEOUT value or set it to "-1" for indefinite waiting.
19 - 24	For internal use only.

115 Object Linking and Embedding - OLE

- What is OLE in the Natural Context? 830
- OLE Documents Support 830
- Embedding and Linking 830
- Visual Editing - In-place Activation 831
- ActiveX Controls Support 832
- OLE Container Control 832
- Attributes, Events and PROCESS GUI Statement Actions 835

What is OLE in the Natural Context?

Natural supports the following OLE technologies:

- OLE Documents
- OLE Visual Editing (In-place Activation)
- ActiveX Controls

If you are new to OLE, it is highly recommended that you first get a basic overview by referring to one of the various sources available. One such source, for example, is the Microsoft Win32 software development kit documentation.

OLE Documents Support

OLE documents is a technology that integrates different Windows applications seamlessly so that the end user can concentrate on the data rather than on handling the different applications. With OLE you can, for example, embed a Word for Windows document in a Natural dialog. Whenever the end user enters the text container to edit the document, the entire Word functionality is available. Thus, the end user does not have to invoke Word.

OLE Documents Support is provided by the Natural dialog element OLE container control.

The OLE documents technology defines container and server applications. A container application is an application that is able to use objects created by a server application. These objects are used by linking or embedding them. In this context, Natural is the container application because the dialog editor provides an OLE container control. A typical server application is Microsoft Word; the Word documents would then be the objects used by Natural.

Embedding and Linking

- Linking means that the content of a document is accessed via a link to an external file. This file is stored in the server's format (for example, a file in *.rtf* format would be stored in a file system outside Natural; the server residing in this external file system would be Microsoft Word).
- Embedding means that the content of a document is maintained in the container application and is stored in the container's internal format. Embedded documents are created
  - either by building them from scratch in the container application;
  - or by loading an external document.

Embedded objects are edited by visual editing („in-place activation“), whereas linked objects must be opened in an extra server window for editing.

Natural provides the dialog element OLE container control for embedding and linking documents. Furthermore, Natural provides actions to save and load embedded documents in internal Natural format. By default, these embedded objects in internal format are stored and retrieved in the %NATGUI_BMP% directory with a default extension of *.neo* (Natural Embedded Object).

To display an embedded object with the OLE container control when the dialog starts

1. Invoke the container control's attribute window.
2. Set the **Type** entry to "Existing OLE Object".
3. Select a file specification in the Name field.

To display an embedded object dynamically at runtime

- Use the `PROCESS GUI` statement action `OLE-READ-FROM-FILE`.

To display a linked object with the OLE container control when the dialog starts

1. Invoke the container control's attribute window.
2. Set the **Type** entry to "OLE Server".
3. In the **Select OLE Server or Document** dialog that comes up, select **Create From File** and select a file specification.

To display a linked object dynamically at runtime

- Assign the file specification of the external document to the attribute `SERVER-OBJECT`.

Visual Editing - In-place Activation

In-place activation means that the end user is able to activate a server application in the container application's window. Such a server application is related to an object embedded in a Natural dialog's OLE container control. The server application is activated by double-clicking on the OLE container control. The Natural dialog's toolbar and menu-bar control are then merged with the server application's menu and toolbar. The dialog now contains toolbar items and menu items that enable you to edit the object with the help of the server's functionality.

ActiveX Controls Support

ActiveX controls support enables the Natural programmer to use the many third-party ActiveX controls inside a Natural dialog. Natural enables you to access the ActiveX controls properties and methods direct and to program the ActiveX controls events.

ActiveX controls support is provided by the Natural dialog element „ActiveX control". For more information, see *Working with ActiveX Controls*.

OLE Container Control

The following topics are covered below:

- Creating an OLE Container Control
- Creating an OLE Container Control in the Dialog Editor
- Creating an OLE Container Dynamically At Runtime
- Clearing or Deleting an OLE Container At Runtime
- OLE Container Controls And The Dialog's Menu Bar
- Other OLE Container Control Functionality

Creating an OLE Container Control

You can create an OLE container control either statically in the dialog editor or dynamically at runtime.

Creating an OLE Container Control in the Dialog Editor

The OLE container control enables you to integrate server applications. You can integrate server applications in the following three ways, as indicated by the **Object Information** group frame, **Type** entry of the OLE container control's attributes window.

- Type: New OLE object. You create an OLE container control that acts as a placeholder for the insertable object. At runtime, your end user can create the embedded object by starting the server application. The embedded object can then be saved as Natural embedded object (*.neo* file).
- Type: Existing OLE object. Your end user changes an existing embedded object in the OLE container control. The embedded object is saved as Natural embedded object (*.neo* file).
- Type: OLE server. You create a native OLE object in your application or you create a link to an external object.

To create an OLE container control in the dialog editor

1. In the dialog editor main menu, choose **Insert**, then **OLE Container**.
2. Draw a rectangle by holding down the right mouse button, dragging the mouse vertically/horizontally and releasing the mouse button.

   An empty OLE container is created.

To display a document in the OLE container when starting the dialog

1. Double-click the OLE container control to invoke the attribute window.
2. In the **Type** selection box, choose **OLE server** for linking an external document. Or choose **Existing OLE object** for reading in an embedded object.
3. Choose the **...** button to select the external or embedded object file.

Creating an OLE Container Dynamically At Runtime

Before you enter the examples in an event-handler section, declare a handle variable for the OLE container control in the local data area of the dialog:

```
01 #OCT-1 HANDLE OF OLECONTAINER
```

Example for creating an OLE container control at runtime and linking an external document:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #OCT-1
   TYPE = OLECONTAINER
   SERVER-OBJECT = 'PICTURE.BMP'
   RECTANGLE-X = 56
   RECTANGLE-Y = 32
   RECTANGLE-W = 336
   RECTANGLE-H = 160
   PARENT = #DLG$WINDOW
   SUPPRESS-CLICK-EVENT = SUPPRESSED
   SUPPRESS-DBL-CLICK-EVENT = SUPPRESSED
   SUPPRESS-CLOSE-EVENT = SUPPRESSED
   SUPPRESS-ACTIVATE-EVENT = SUPPRESSED
   SUPPRESS-CHANGE-EVENT = SUPPRESSED
END-PARAMETERS GIVING *ERROR
```

Example for creating an OLE container control at runtime and embedding a Natural embedded object:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #OCT-1
   TYPE = OLECONTAINER
   EMBEDDED-OBJECT = 'SLIDE.NEO'
   RECTANGLE-X = 56
   RECTANGLE-Y = 32
   RECTANGLE-W = 336
   RECTANGLE-H = 160
   PARENT = #DLG$WINDOW
   SUPPRESS-CLICK-EVENT = SUPPRESSED
   SUPPRESS-DBL-CLICK-EVENT = SUPPRESSED
   SUPPRESS-CLOSE-EVENT = SUPPRESSED
   SUPPRESS-ACTIVATE-EVENT = SUPPRESSED
   SUPPRESS-CHANGE-EVENT = SUPPRESSED
   END-PARAMETERS GIVING *ERROR
```

Clearing or Deleting an OLE Container At Runtime

This section contains examples for clearing and deleting an OLE container at runtime.

Before you enter the examples in an event-handler section, declare a handle variable for the OLE container control in the local data area of the dialog:

```
01 #OCT-1 HANDLE OF OLECONTAINER
```

Example for clearing (removing the document of) the OLE container control:

```
PROCESS GUI ACTION CLEAR WITH #OCT-1
```

Example for deleting the OLE container control:

```
PROCESS GUI ACTION DELETE WITH #OCT-1
```

OLE Container Controls And The Dialog's Menu Bar

The menu item attribute MENU-ITEM-OLE can have four different values which detemine if and where the menu item in question is displayed during in-place activation of a server.

The menu item attribute MENU-ITEM-TYPE also has the value MT-OBJECTVERBS. This enables you to have the OLE container control display the available server actions (command verbs) in this menu item.

Other OLE Container Control Functionality

While a document is displayed in an OLE container control, the end user has the possibility to activate the default command verb of the server by double-clicking inside the OLE container control's rectangle. This is equivalent to executing the PROCESS GUI statement action OLE-ACTIVATE. Furthermore, the end user can select a server command verb by displaying a popup menu. You display this popup menu by holding down the right mouse button inside the OLE container. Then you select the desired command verb and release the mouse button.

If the MODIFIABLE attribute of an OLE container control is set to FALSE, a double-click on the container does not start the default command verb of the server and holding down the right mouse button does not show the popup menu with the available server command verbs (see also ***Executing Standardized Procedures***).

During visual editing (in-place activation), the server uses the Natural dialog for the editing of the document. The server does its work as a task on its own and the Natural processing continues. Thus, it is possible to execute event code and, for example, to limit the visual editing to a certain time by specifying PROCESS GUI ACTION OLE-DEACTIVATE, WITH #OCT-1 in a timer's event section (see also ***Executing Standardized Procedures***).

Attributes, Events and PROCESS GUI Statement Actions

The following sections list all the attributes, events and PROCESS GUI statement actions that apply specifically to the OLE container control.

Attributes

The OLE-specific attributes provided with the OLE container control are:

- EMBEDDED-OBJECT
- ICONIZED
- OBJECT-SIZE
- SERVER-OBJECT

- SERVER-PROGID
- SUPPRESS-ACTIVATE-EVENT
- SUPPRESS-CLOSE-EVENT
- ZOOM-FACTOR

Event

This OLE-specific event occurs when a server application is activated:

- Activate event

PROCESS GUI Statement Actions

The OLE-specific PROCESS GUI statement actions provided with the OLE container control are:

- OLE-ACTIVATE
- OLE-DEACTIVATE
- OLE-GET-DATA
- OLE-INSERT-OBJECT
- OLE-READ-FROM-FILE
- OLE-SAVE-TO-FILE
- OLE-SET-DATA

116 Results Interface

- Zweck des Results Interfaces ... 838
- Auf das Results-Fenster zugreifen ... 839
- Registerkarten ... 839
- Bilder ... 840
- Kontextmenüs ... 840
- Befehle ... 841
- Spalten ... 841
- Zeilen ... 842
- Daten ... 842
- Markierungen ... 842

Zweck des Results Interfaces

Mit dem Results Interface können Daten im Results-Fenster (Ergebnisfenster) von Natural Studio angezeigt werden. Siehe auch *Results-Fenster* in der Dokumentation *Natural Studio benutzen*.

Anmerkung: Die Ergebnisse der Menübefehle **Find Objects** und **Cat All** werden vom Results Interface nicht beeinflusst.

Das Aussehen und die Benutzung einer Registerkarte im Results-Fenster kann mit Hilfe von Programmierschnittstellen (APIs) definiert werden. In der Regel wird eine detaillierte Ansicht mit Spalten und Zeilen benutzt.

Für jeden Eintrag kann ein Kontextmenü erstellt werden. Somit kann ein Eintrag nach der Anzeige einer benutzerdefinierten Registerkarte für weitere Verarbeitungsschritte verwendet werden.

Diese Weiterverarbeitung muss in zwei Programmen definiert werden:

1. In einem Update-Command-Handler bevor das Kontextmenü angezeigt wird.
2. In einem Command-Handler, falls ein Eintrag markiert ist.

Die Programmierschnittstellen für das Results Interface sind `USR5001N` - `USR5017N`. Sie befinden sich in der Library `SYSEXT`.

Ein Beispiel der verschiedenen Funktionen finden Sie in `USR5001P`, wobei der Update-Command-Handler in `USR5001A` und der Command-Handler in `USR5001B` zu finden ist.

Anmerkungen:

1. Mit dem Results Interface ist es nicht möglich, Änderungen an den vordefinierten Registerkarten (zum Beispiel, an den Registerkarten mit den Namen **Find Objects** und **Cat All**) vorzunehmen.
2. Das Results-Fenster und das Results Interface können nur von Natural Studio aus aufgerufen werden.

Auf das Results-Fenster zugreifen

Mit der folgenden Programmierschnittstelle können Sie auf das Results-Fenster zugreifen.

API	Funktionalität
USR5001N	Schaltet das Results-Fenster ein und aus. Überprüft die Sichtbarkeit des Results-Fensters.

Registerkarten

Mit den unten aufgeführten Programmierschnittstellen kann das allgemeine Layout einer Registerkarte definiert werden.

Eine Registerkarte kann alles oder nur eines von Folgendem enthalten:

- Kontrollkästchen
- Full-Row-Selektion
- Single-Row-Selektion
- Bilder

Eine Registerkarte kann mit den folgenden Attributen definiert werden:

- Layout der Ansicht (große/kleine Icons, Liste oder detaillierte Ansicht).
- Unterschiedliche Verwendungen (Kontrollkästchen, Bilder, Gitterlinien, Full-Row-Selektion oder Single-Row-Selektion, andere Ansicht).
- Layout des Registerkartennamens (Text, Bitmap oder Icon).

API	Funktionalität
USR5004N	Layout für eine Registerkarte hinzufügen, ersetzen, löschen oder verwalten.
USR5005N	Aktive Registerkarte setzen und holen. Aktive Registerkarte festlegen und den Fokus auf diese Registerkarte setzen.

Bilder

Mit der folgenden Programmierschnittstelle können Sie Bitmaps (*.*bmp*) und Icons (*.*ico*) für eine bereits definierte Registerkarte angeben.

API	Funktionalität
USR5002N	Auf einer Registerkarte Bitmaps und Icons hinzufügen oder löschen.

Kontextmenüs

Mit den folgenden Programmierschnittstellen können Sie benutzerdefinierte Kontextmenüs angeben.

API	Funktionalität
USR5003N	Auf einer Registerkarte Kontextmenüs hinzufügen, entfernen und löschen.
USR5007N	Den Status (markiert/aktiviert) eines Befehls in einem Kontextmenü setzen und holen.

Die Hierarchie des Kontextmenüs muss manuell definiert werden.

Die folgenden Array-Bestandteile können definiert werden:

Array-Bestandteil	Wert	Beschreibung
Typ	1 bis 4	1 - Kontextmenübearbeitung. 2 - Trennlinie. 3 - Beginn eines Untermenüs. 4 - Ende eines Untermenüs.
Befehlskennzeichen	1 bis 255	Frei wählbare Zahl für einen bestimmten Befehl im Kontextmenü (wird im Command-Handler benutzt).
Bezeichnung	Alphanumerischer Text	Text für Kontextmenübefehle vom Typ 1 und 3. Ein Text für die Statusleiste kann mit "H'0'A" abgegrenzt werden.
Bild	Image-Handle	Handle für ein bereits definiertes Bild (Bitmap oder Icon). Das Bild wird vor dem Text für den Kontextmenübefehl platziert.

Befehle

Ein Programm kann als Update-Command-Handler oder als Command-Handler zugewiesen werden.

Benutzerdaten können im internen Arbeitsbereich des Command-Handlers gespeichert oder wieder hergestellt werden.

Beispiel: Handles für Registerkarten.

Die folgenden Programmierschnittstellen stehen zur Verfügung:

API	Funktionalität
USR5006N	Update-Command-Handler und Command-Handler definieren.
USR5016N	Daten für den Command-Handler-Arbeitsbereich setzen und holen.

Spalten

Mit den folgenden Programmierschnittstellen kann das allgemeine Spalten-Layout definiert werden.

Eine Spalte kann alles oder nur eines von Folgendem enthalten:

- Titel
- Breite
- Datenposition
- Spaltensortierung

Außerdem können die Standardbreite und die angegebene Breite einer Spalte individual gesetzt werden.

API	Funktionalität
USR5008N	Auf einer Registerkarte Spalten hinzufügen, einfügen und löschen.
USR5009N	Anzahl der Spalten zählen.
USR5010N	Standardspaltenbreite und Breite der angegebenen Spalten setzen und holen.

Zeilen

Mit den folgenden Programmierschnittstellen können die Zeilen definiert werden, die Bilder und Kontextmenüs enthalten.

API	Funktionalität
USR5009N	Anzahl der Zeilen zählen.
USR5011N	Auf einer Registerkarte Zeilen hinzufügen, einfügen und löschen.
USR5015N	Eine Zeile kann in den sichtbaren Bereich des Result-Fensters gerollt werden.

Daten

Mit den folgenden Programmierschnittstellen können Benutzerdaten in definierte Spalten/Zeilen geschrieben werden.

Wenn auf einer Registerkarte Kontrollkästchen definiert wurden, können sie in jeder Zeile aktiviert/deaktiviert werden.

API	Funktionalität
USR5012N	Daten für eine Registerkarte setzen und holen.
USR5013N	Aktivierungsstatus einer Zeile setzen und holen.

Markierungen

Mit den folgenden Programmierschnittstellen können Zeilen individuell markiert werden.

API	Funktionalität
USR5014N	■ Markierte Zeilen setzen, zurücksetzen und holen. ■ Anzahl der markierten Zeilen zählen. ■ Zeilenmarkierung setzen und zurücksetzen.
USR5015N	Zeile mit dem Fokus setzen und holen.
USR5017N	Markierte Zeilen in die Zwischenablage kopieren.

117 Gestaltung von zeichenbasierten Benutzeroberflächen von Anwendungen

Die Benutzeroberfläche einer Anwendung, d.h. die Art und Weise, in der sich eine Anwendung dem Benutzer präsentiert, ist von entscheidender Bedeutung beim Erstellen einer Anwendung.

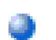
Dieser Teil beschreibt die verschiedenen Möglichkeiten, die Natural bietet, um zeichenorientierte Benutzeroberflächen zu erstellen, die ein einheitliches Erscheinungsbild haben und eine einfache, jedoch flexible Benutzerführung bieten.

Anmerkung: Informationen über das Erstellen von grafischen Benutzeroberflächen (GUI) finden Sie unter *Introduction to Event-Driven Programming*.

Beim Entwurf von Benutzeroberflächen spielen Standards und Standardisierung eine wichtige Rolle.

Mit Natural ist es möglich, dem Benutzer eine einheitliche Benutzeroberfläche zu präsentieren, auch über Hardware- und Betriebssystemgrenzen hinaus.

Zum Design einer solchen Oberfläche gehören der allgemeine Bildschirmaufbau (Informations-, Daten- und Meldezeilenbereich), die Funktionstastenbelegung und der Aufbau von Fenstern.

 Bildschirmgestaltung — Definition des allgemeinen Layouts von Bildschirmen

Dialog-Gestaltung — Gestaltung von Benutzungsschnittstellen

118 Bildschirmgestaltung

- Steuerung der Meldungszeile — Terminalkommando %M 846
- Zuweisen von Farben zu Feldern — Terminalkommando %= 846
- Statistikzeile/Infoline — Terminalkommando %X 848
- Fenster 848
- Standard-/Dynamische Layout-Maps 854
- Mehrsprachige Benutzeroberflächen 855
- Kenntnisabhängige Benutzeroberflächen (Expertenmodus) 860

Dieses Kapitel beschreibt die Möglichkeiten zur Gestaltung des Bildschirm-Layouts.

Steuerung der Meldungszeile — Terminalkommando %M

Mit dem Terminalkommando `%M` geben Sie an, wie und wo die Natural-Meldungszeile angezeigt werden soll.

Im folgenden finden Sie Informationen zu:

- Positionierung der Meldungszeile
- Farbe der Meldungszeile

Positionierung der Meldungszeile

%MB
Die Meldungszeile wird am unteren Bildschirmrand angezeigt:

%MT
Die Meldungszeile wird am oberen Bildschirmrand angezeigt:

Weitere Optionen zur Positionierung der Meldungszeile sind im Abschnitt *%M - Steuerung der Meldungszeile* in der *Terminalkommandos*-Dokumentation beschrieben.

Farbe der Meldungszeile

%M=*`color-code`*
Die Meldungszeile wird in der angegebenen Farbe angezeigt (eine Beschreibung der Farbcodes finden Sie unter Session-Parameter `CD` in der *Parameter-Referenz*-Dokumentation.

Zuweisen von Farben zu Feldern — Terminalkommando %=

Mit dem Terminalkommando `%=` können Sie bestimmten Feldern bestimmte Farben zuweisen, und zwar für Programme, die ursprünglich ohne Berücksichtigung von Farbgebung geschrieben wurden. Sie geben einen Feldtyp und/oder ein Feldattribut an sowie eine Farbe. Alle Felder/Texte dieses Typs/Attributs werden dann in dieser Farbe angezeigt.

Außerdem können Sie bestehende Farbzuweisungen ändern, falls bereits vordefinierte Farbgebungen ungeeignet sind.

Darüber hinaus können Sie das Terminalkommando `%=` in den Natural-Editoren benutzen, um Farben dynamisch zuordnen, z.B. beim Erstellen einer Maske (Map).

Codes	Beschreibung
leer	Bestehende Farbzuweisungen werden gelöscht.
`F`	Neu definierte Farbzuweisungen gelten statt denen des Programms.
`N`	Im Programm definierte Farbzuweisungen behalten ihre Gültigkeit.
`O`	Ausgabefeld (Output).
`M`	Modifizierbares Feld (Aus- und Eingabe).
`T`	Textkonstante.
`B`	Blinkend.
`C`	Kursiv.
`D`	Standard (Default).
`I`	Intensiviert.
`U`	Unterstrichen.
`V`	Invers.
`BG`	Bildschirmhintergrund (Background).
`BL`	Blau.
`GR`	Grün.
`NE`	Neutral.
`PI`	Rosa (Pink).
`RE`	Rot (Red).
`TU`	Türkis.
`YE`	Gelb (Yellow).

Beispiel:

```
%=TI=RE,OB=YE
```

Dieses Beispiel ordnet die Farbe Rot allen intensivierten Text-Feldern und Gelb allen blinkenden Ausgabefeldern zu.

Statistikzeile/Infoline — Terminalkommando %X

Dieses Terminalkommando steuert die Anzeige der Natural-Infoline.

Weitere Informationen finden Sie in der Beschreibung des Terminalkommandos `%X` in der *Terminalkommandos*-Dokumentation.

Fenster

Im folgenden finden Sie Informationen zu:

- Was ist ein Fenster?
- DEFINE WINDOW-Statement
- INPUT WINDOW-Statement

Was ist ein Fenster?

Ein *Fenster* ist jener, von einem Programm aufgebaute Abschnitt einer logischen Seite, der auf dem Terminal-Bildschirm angezeigt wird.

Eine *logische Seite* ist der Ausgabebereich für Natural; mit anderen Worten enthält die logische Seite den/die vom Natural-Programm für die Anzeige erzeugte/n Report/Map. Diese logische Seite kann breiter als der physische Bildschirm sein.

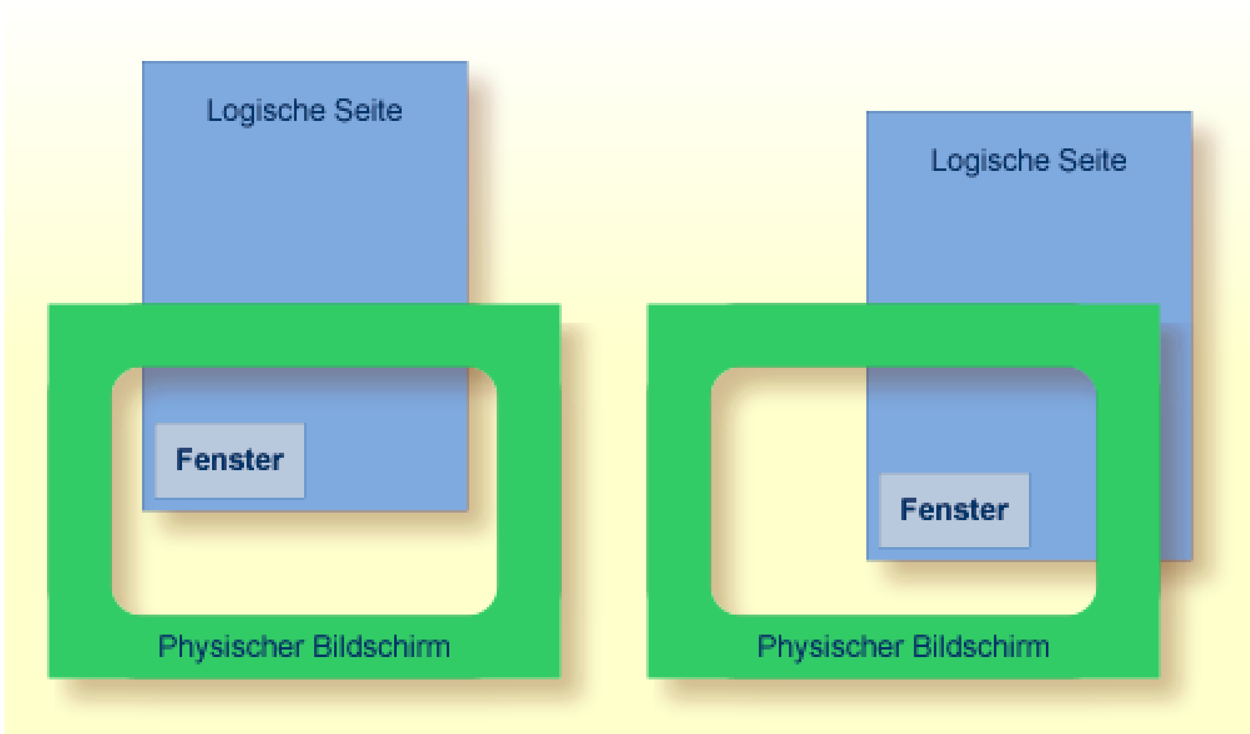
Es ist immer ein Fenster vorhanden, auch wenn dessen Vorhandensein Ihnen nicht bewusst sein mag. Wenn es (durch ein `DEFINE WINDOW`-Statement) nicht anders angegeben ist, ist die Größe des Fensters mit der physischen Größe Ihres Terminal-Bildschirms identisch.

Sie können ein Fenster auf zwei Arten handhaben:

- Sie können die Größe und Position des Fensters auf dem *physischen Bildschirm* steuern.
- Sie können die Position des Fensters auf der *logischen Seite* steuern.

Positionierung auf dem physischen Bildschirm

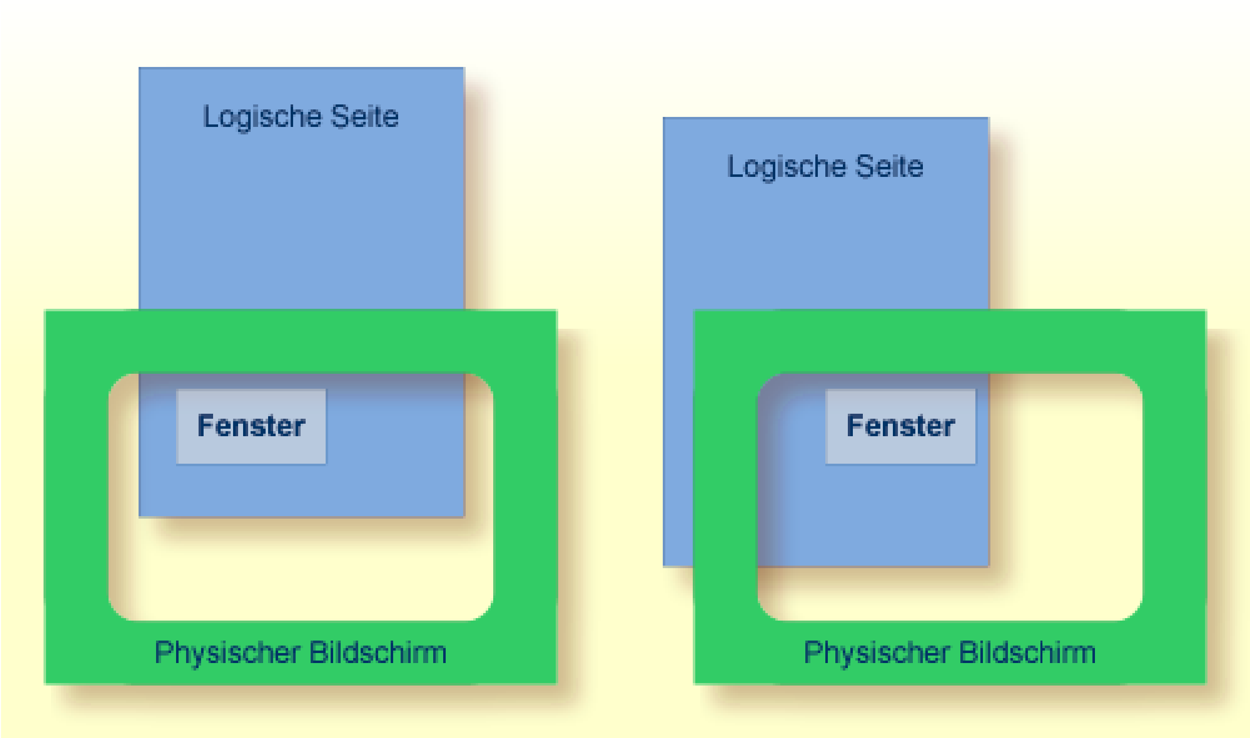
Die Abbildung unten veranschaulicht die Positionierung des Fensters auf dem physischen Bildschirm. Beachten Sie, dass in beiden Fällen der gleiche Ausschnitt der logischen Seite angezeigt wird; es wird lediglich die Position des Fensters auf dem physischen Bildschirm geändert.



Positionierung auf der logischen Seite

Die Abbildung unten veranschaulicht die Positionierung des Fensters auf der logischen Seite.

Wenn Sie die Position des Fensters auf der *logischen Seite* verändern, bleiben Position und Größe des Fensters auf dem *physischen Bildschirm* gleich; d.h. das Fenster wird nicht über der logischen Seite bewegt, sondern die logische Seite wird sozusagen „unter“ dem Fenster verschoben.



DEFINE WINDOW-Statement

Das DEFINE WINDOW-Statement dient dazu, die Größe, Position und Attribute eines Bildschirmfensters auf dem *physischen Bildschirm* zu definieren.

Mit einem DEFINE WINDOW-Statement wird ein Fenster nicht aktiviert; dies geschieht mit einem SET WINDOW-Statement oder der WINDOW-Klausel eines INPUT-Statements.

Das DEFINE WINDOW-Statement bietet verschiedene Optionen. Diese werden anhand des nachstehenden Beispiels erläutert.

Das folgende Programm definiert ein Fenster auf dem physischen Bildschirm.

```
** Example 'WINDX01': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST
       SIZE 5*25
       BASE 5/40
       TITLE 'Sample Window'
       CONTROL WINDOW
       FRAMED POSITION SYMBOL BOTTOM LEFT
```

```
*
INPUT WINDOW='TEST' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST2'
  FETCH 'WINDX02'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

Der Window-Name identifiziert das Fenster. Der Name darf bis zu 32 Stellen lang sein. Für Fensternamen gelten die gleichen Namenskonventionen wie für Benutzervariablen. Hier lautet der Name `TEST`.

Mit der `SIZE`-Klausel bestimmen Sie die Größe des Fensters. In dem Beispiel ist das Fenster 5 Zeilen hoch und 25 Spalten (Stellen) breit.

Mit der `BASE`-Klausel bestimmen Sie die Position des Fensters auf dem physischen Bildschirm. In dem Beispiel ist die obere linke Ecke des Fensters auf Zeile 5, Spalte 40 positioniert.

Mit der `TITLE`-Klausel können Sie eine Überschrift für das Fenster angeben. Die angegebene Überschrift wird zentriert in der oberen Rahmenzeile des Fensters angezeigt (natürlich nur, wenn ein Rahmen für das Fenster definiert ist).

Mit der `CONTROL`-Klausel können sie festlegen, ob die PF-Tastenzeile, die Meldungs- und die Statistikzeile in dem Fenster oder auf dem vollen physischen Bildschirm angezeigt werden. In diesem Fall bewirkt `CONTROL WINDOW`, dass die Meldungszeile im Fenster angezeigt wird. `CONTROL SCREEN` hingegen bewirkt, dass die Zeilen auf dem vollen physischen Bildschirm außerhalb des Fenster angezeigt werden. Wenn Sie die `CONTROL`-Klausel weglassen, gilt standardmäßig `CONTROL WINDOW`.

Mit der `FRAMED`-Option geben Sie an, dass das Fenster eingerahmt werden soll.

Dieser Rahmen ist dann cursor-sensitiv. Sie können gegebenenfalls im Fenster vor, zurück, nach rechts oder links blättern, indem Sie den Cursor einfach auf das entsprechende Symbol `<`, `-`, `+` oder `>` (vgl. `POSITION`-Klausel weiter unten) platzieren und EINGABE drücken. Anders ausgedrückt, bewegen Sie damit die logische Seite unter dem Fenster auf dem physischen Bildschirm. Werden keine Symbole angezeigt, können Sie vor- und zurückblättern, indem Sie den Cursor in die obere (zum Zurückblättern) bzw. untere (zum Vorblättern) Rahmenzeile platzieren und EINGABE drücken.

Mit der `POSITION`-Klausel der `FRAMED`-Option bestimmen Sie, dass Informationen über die Position des Fensters auf der logischen Seite in dem Fensterrahmen angezeigt werden. Diese Positionsan-

gaben werden nur angezeigt, wenn die logische Seite größer ist als das Fenster; sonst wird die POSITION-Klausel ignoriert. Die Positionsangaben geben an, in welche Richtungen die logische Seite nach oben, unten, links und rechts über das aktuelle Fenster hinausgeht.

Wird keine POSITION-Klausel angegeben, gilt standardmäßig POSITION SYMBOL TOP RIGHT.

POSITION SYMBOL bewirkt, dass die Positionsangaben als Symbole More: < - + > angezeigt werden. Die Angaben werden in der oberen und/oder unteren Rahmenzeile angezeigt.

TOP/BOTTOM bestimmt, in welcher Rahmenzeile (oben oder unten) die Angaben erscheinen sollen.

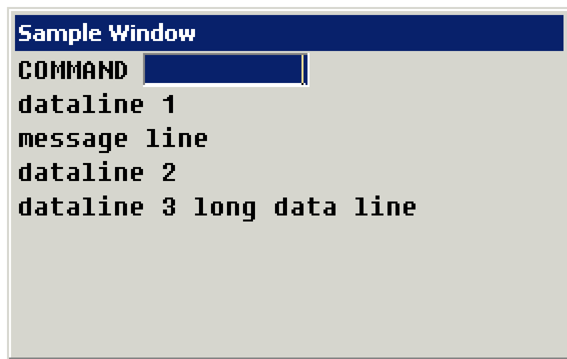
LEFT/RIGHT bestimmt, ob die Positionsangaben im linken oder rechten Teil der Rahmenzeile angezeigt werden.

INPUT WINDOW-Statement

Das INPUT WINDOW-Statement aktiviert das in dem DEFINE WINDOW-Statement definierte Fenster. In dem Beispiel wird das Fenster TEST aktiviert. Wollen Sie Daten in einem Fenster ausgeben (z.B. mit einem WRITE-Statement), benutzen Sie das SET WINDOW-Statement.

Wenn das obige Programm ausgeführt wird, wird das Fenster mit einem Eingabefeld COMMAND angezeigt. Mit dem Session-Parameter AD legen Sie fest, dass der Wert des Feldes intensiviert dargestellt und ein Unterstrich als Füllzeichen benutzt wird.

Ausgabe des Programms WINDX01:



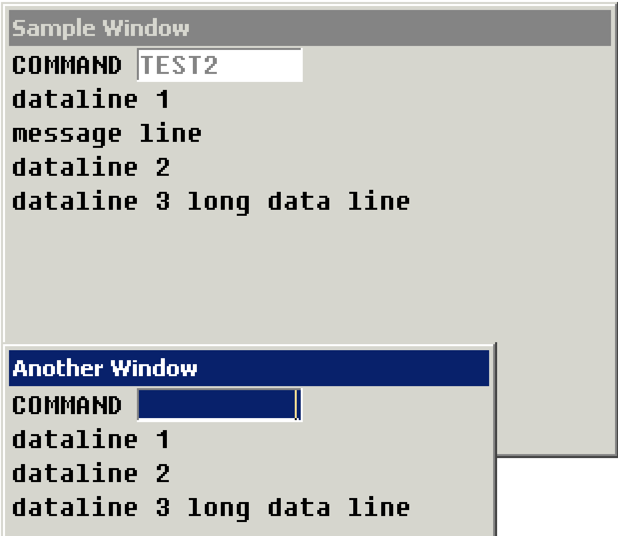
Mehrere Fenster

Sie können mehrere Fenster öffnen. Allerdings ist jeweils nur ein Natural-Fenster aktiv, und zwar das letzte. Vorherige Fenster mögen auf dem Bildschirm noch sichtbar sein, sind aber nicht mehr aktiv und werden von Natural ignoriert. Sie können Eingaben nur im jeweils letzten Fenster machen. Sollte der Platz hierzu nicht ausreichen, müssen Sie das Fenster vorher entsprechend vergrößern.

Wird `TEST2` in das `COMMAND`-Feld eingegeben, wird das Programm `WINDX02` ausgeführt.

```
** Example 'WINDX02': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST2
       SIZE 5*30
       BASE 15/40
       TITLE 'Another Window'
       CONTROL SCREEN
       FRAMED POSITION SYMBOL BOTTOM LEFT
*
INPUT WINDOW='TEST2' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST'
  FETCH 'WINDX01'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

Ein zweites Fenster wird nun geöffnet. Das andere Fenster ist zwar noch sichtbar, jedoch inaktiv.



Beachten Sie, dass die Meldungszeile (message line) für das neue Fenster außerhalb des Fensters (unten auf dem physischen Bildschirm) angezeigt wird. Dies wurde mit der `CONTROL SCREEN`-Klausel im Programm `WINDX02` definiert.

Weitere Einzelheiten zu den Statements `DEFINE WINDOW`, `INPUT WINDOW` und `SET WINDOW` finden Sie in den entsprechenden Beschreibungen in der *Statements*-Dokumentation.

Standard-/Dynamische Layout-Maps

Standard-Layout-Maps

Im Map-Editor kann ein *Standard-Layout* definiert werden. Wird dieses Layout bei der Erstellung aller Maps verwendet, so wird gewährleistet, dass die gesamte Anwendung ein einheitliches Erscheinungsbild aufweist.

Wenn eine Map, die ein Standard-Layout referenziert, initialisiert wird, wird dieses Layout zum festen Bestandteil der Map. Falls das Standard-Layout nachträglich geändert wird, bedeutet dies allerdings, dass alle Maps neu katalogisiert werden müssen, damit die Änderungen greifen.

Dynamische Layout-Maps

Im Gegensatz zum Standard-Layout, wird ein *dynamisches Layout* nicht zum festen Bestandteil einer sich darauf beziehenden Map; vielmehr wird das Layout jeweils zur Laufzeit generiert.

Wenn Sie also auf dem Define Map Settings For MAP-Schirm im Map-Editor die Layout-Map als „dynamisch" definieren, werden alle Änderungen der Layout-Map automatisch auch bei allen Maps, die sich darauf beziehen, durchgeführt. Die Maps müssen nicht neu katalogisiert werden.

Weitere Einzelheiten über Layout-Maps finden Sie in der *Map Editor* in the *Editors*-Dokumentation.

Mehrsprachige Benutzeroberflächen

Mit Natural können Sie mehrsprachige Anwendungen für den internationalen Einsatz erstellen.

Maps, Helproutinen, Fehlermeldungen, Programme, Functions,Subprogramme und Copycodes können in bis zu 60 verschiedenen Sprachen (inklusive Sprachen, die einen Doppelbyte-Zeichensatz benutzen) definiert werden.

Im folgenden finden Sie Informationen zu:

- Sprachcodes
- Definition der Sprache eines Natural-Objektes
- Definition der Benutzersprache
- Referenzieren von mehrsprachigen Objekten
- Programme
- Fehlermeldungen
- Editiermasken für Datums- und Uhrzeitfelder

Sprachcodes

In Natural hat jede Sprache einen *Sprachcode* (von `1` bis `60`). Die folgende Tabelle ist ein Auszug der Gesamttabelle der Sprachcodes.

Eine vollständige Aufstellung der Sprachcodes finden Sie in der Beschreibung der Systemvariablen `*LANGUAGE` in der *Systemvariablen*-Dokumentation.

Sprachcode	Sprache	Mapcode in Objekt-Namen
1	Englisch	1
2	Deutsch	2
3	Französisch	3
4	Spanisch	4
5	Italienisch	5
6	Niederländisch	6
7	Türkisch	7
8	Dänisch	8
9	Norwegisch	9
10	Albanisch	A
11	Portugiesisch	B

Der Sprachcode (linke Spalte) ist der Code, der in der Systemvariable `*LANGUAGE` enthalten ist. Dieser Code wird von Natural intern benutzt. Es ist der Code, den Sie zur Definition der Benutzersprache benutzen (siehe ***Definition der Benutzersprache*** weiter unten). Der Code, den Sie zur Identifikation der Sprache eines Natural-Objekts angeben, ist der *Mapcode* in der rechten Spalte der Tabelle.

Beispiel:

Der Sprachcode für Portugiesisch ist `11`. Der Code, den Sie beim Katalogisieren eines portugiesischen Natural-Objekts angeben, ist `B`.

Definition der Sprache eines Natural-Objektes

Sie definieren die Sprache eines Natural-Objektes (Map, Helproutine, Programm, Function,Subprogramm oder Copycode), indem Sie den entsprechenden Mapcode dem Objektnamen hinzufügen. Bis auf diesen Mapcode muss der Objektname identisch für alle Sprachen sein.

Beim folgenden Beispiel wurden zwei Maps erzeugt, und zwar eine englische und eine deutsche. Um die Sprachen der Maps zu identifizieren, wurde der entsprechende Mapcode jeweils den Mapnamen hinzugefügt.

Beispiel für Mapnamen bei einer mehrsprachigen Anwendung:

`DEMO1` = englische Map (Mapcode 1)

`DEMO2` = deutsche Map (Mapcode 2)

Definition von Sprachen mit alphabetischen Mapcodes

Mapcodes können im Bereich `1-9`, `A-Z` oder `a-y` liegen. Die alphabetischen Mapcodes bedürfen einer besonderen Handhabung.

Normalerweise ist es nicht möglich, ein Objekt zu katalogisieren, das einen Kleinbuchstaben im Namen hat — alle Buchstaben werden automatisch in Großbuchstaben umgewandelt.

Genau dies ist aber erforderlich, wenn Sie z.B. ein Objekt als japanisch (Kanji) definieren wollen. Japanisch hat den Sprachcode `59` und den Mapcode `x`.

Um ein solches Objekt zu katalogisieren, müssen Sie zuerst den Sprachcode (in diesem Fall Sprachcode `59`) richtig setzen, indem Sie das Terminalkommando `%L=`*nn* (*nn* entspricht dem Sprachcode) benutzen.

Jetzt können Sie das Objekt katalogisieren, wobei Sie das Und-Zeichen (&) anstelle des eigentlichen Mapcodes im Objektnamen benutzen. Um eine japanische Version der Map `DEMO` zu erhalten, katalogisieren Sie die Map also unter dem Namen `DEMO&`.

Wenn Sie jetzt in der Liste der Natural-Objekte nachschauen, wird die Map richtigerweise als `DEMOx` aufgelistet.

Sie können Objekte mit Sprachcode `1` bis `9` und `A` bis `Z` direkt katalogisieren, d.h. ohne die `&`-Notation.

Im nachfolgenden Beispiel sehen Sie die drei Maps `DEMO1`, `DEMO2` und `DEMOx`. Um die Map `DEMOx` zu löschen, benutzen Sie die gleiche Technik wie bei der Definition des Mapnamens, d.h. Sie setzen zuerst die richtige Sprache mit dem Terminalkommando `%L=59`, dann bestätigen Sie den Löschvorgang mit der `&`-Notation (`DEMO&`).

Definition der Benutzersprache

Sie können pro Benutzer bestimmen, welche Sprache (wie in der Systemvariablen `*LANGUAGE` definiert) benutzt wird, und zwar mit dem Profilparameter `ULANG` (der in der *Parameter-Referenz*-Dokumentation beschrieben ist) oder mit dem Terminalkommando `%L=`*nn* (wobei *nn* der Sprachcode ist).

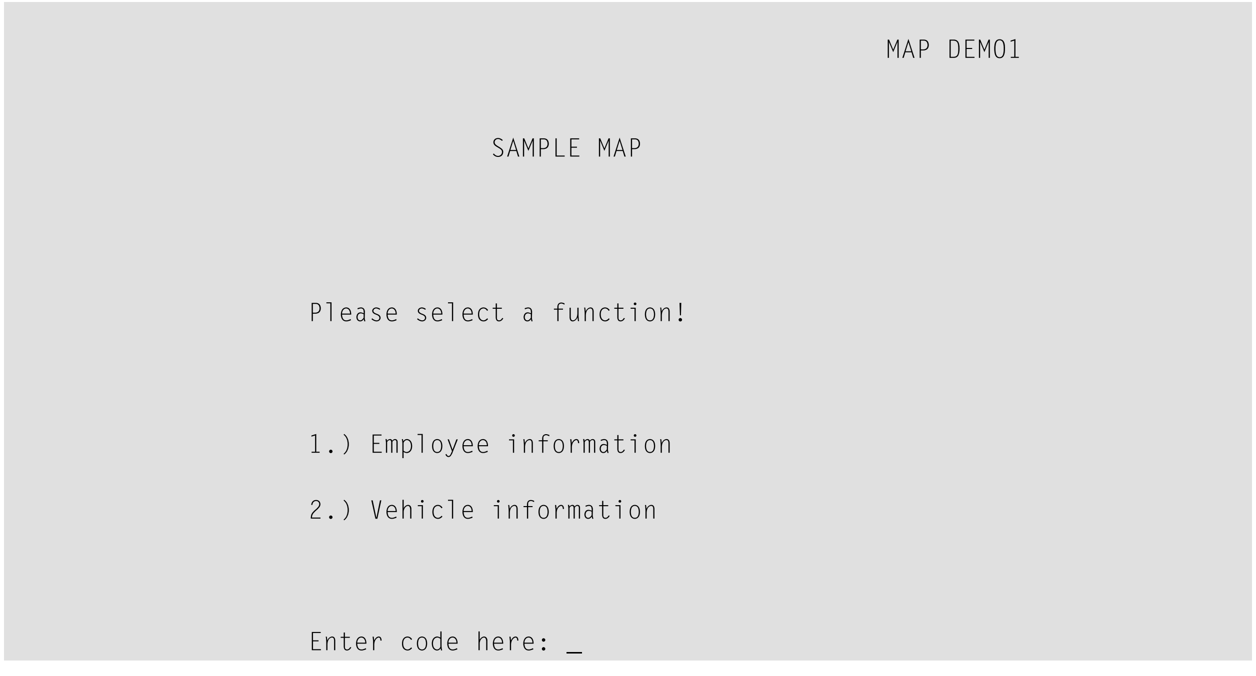
Referenzieren von mehrsprachigen Objekten

Um in einem Programm mehrsprachige Objekte zu referenzieren, benutzen Sie das `&`-Zeichen im Namen des Objektes.

Das Programm unten benutzt die zwei Maps `DEMO1` und `DEMO2`. Das `&`-Zeichen am Ende des Mapnamens steht anstelle des Mapcodes und bedeutet, dass die Map mit der Sprache, die dem Wert in der Systemvariable `*LANGUAGE` entspricht, benutzt werden soll.

```
DEFINE DATA LOCAL
1 PERSONNEL VIEW OF EMPLOYEES
  2 NAME (A20)
  2 PERSONNEL-ID (A8)
1 CAR VIEW OF VEHICLES
   2 REG-NUM (A15)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'DEMO&' /* <--- INVOKE MAP WITH CURRENT LANGUAGE CODE
...
```

Wird dieses Programm ausgeführt, so wird die englische Map (`DEMO1`) ausgegeben, denn der Wert von `*LANGUAGE` ist `1` = englisch.



Im Beispiel unten wird der Sprachcode auf `2` = deutsch umgesetzt und zwar mit dem Terminalkommando `%L=2`.

Wird das Programm nun ausgeführt, wird die deutsche Map (DEMO2) ausgegeben.

```
                              BEISPIEL-MAP



                  Bitte wählen Sie eine Funktion!



                  1.) Mitarbeiterdaten

                  2.) Fahrzeugdaten



               Code hier eingeben: _
```

Programme

Bei manchen Anwendungen kann es nützlich sein, mehrsprachige Programme zu definieren. Ein Fakturierungsprogramm könnte z.B. verschiedene Unterprogramme benutzen, um gewisse steuerliche Aspekte zu berücksichtigen, je nachdem in welchem Land die Rechnung erstellt werden soll.

Mehrsprachige Programme werden genauso definiert wie Maps (siehe Beschreibung oben).

Fehlermeldungen

Mit der Natural-Utility SYSERR können Sie die Natural-Fehlermeldungen in bis zu 60 Sprachen übersetzen. Sie können aber auch eigene Fehlermeldungen erstellen.

Die Sprache der ausgegebenen Fehlermeldungen wird durch den Inhalt der Systemvariable *LANGUAGE bestimmt.

Einzelheiten über Fehlermeldungen finden Sie in der *SYSERR Utility* in der *Utilities*-Dokumentation.

Editiermasken für Datums- und Uhrzeitfelder

Die Sprache für Datums- und Uhrzeitfelder, die mit Editiermasken definiert wurden, wird auch durch die Systemvariable `*LANGUAGE` festgelegt.

Weitere Einzelheiten zu Editiermasken entnehmen Sie dem Session-Parameter `EM` in der *Parameter Reference*.

Kenntnisabhängige Benutzeroberflächen (Expertenmodus)

Es kann sinnvoll sein, Benutzern mit unterschiedlicher Erfahrung bei der Benutzung derselben Anwendung verschiedene Maps mit unterschiedlichem Informationsgehalt zu bieten.

Ist Ihre Anwendung *nicht* für den internationalen Einsatz bestimmt, können Sie die gleichen Techniken, die zur Unterstützung von mehrsprachigen Anwendungen benutzt werden, auch zur Definition von unterschiedlich detaillierten Maps verwenden.

Sie können z.B. Sprachcode `1` als Kenntnisstufe 1 = Kenntnisstand eines Anfängers, und Sprachcode `2` als Kenntnisstufe 2 = Kenntnisstand eines fortgeschrittenen Benutzers definieren. Die Anwendung dieser einfachen Technik wird im Folgenden veranschaulicht.

Die folgende Map (`PERS1`) enthält ausführliche Anweisungen, die dem Endbenutzer sagen, wie eine Funktion ausgewählt wird. Die Informationen sind sehr detailliert. Der Name dieser Map enthält den Mapcode `1`.

```
                                                          MAP PERS1

                                  SAMPLE MAP

                    Please select a function


                    1.) Employee information  _

                    2.) Vehicle information   _

                        Enter code:  _

  To select a function, do one of the following:

  - place the cursor on the input field next to desired function and press ENTER
  - mark the input field next to desired function with an X and press ENTER
  - enter the desired function code (1 or 2) in the 'Enter code' field and press
    ENTER
```

Die gleiche Map - aber ohne die ausführlichen Anweisungen - wird unter dem gleichen Namen, aber mit Mapcode 2 gespeichert.

```
                                                     MAP PERS2


                                   SAMPLE MAP

                        Please select a function


                        1.) Employee information  _

                        2.) Vehicle information   _

                            Enter code:  _
```

In dem obigen Beispiel wird die Map mit den vollständigen Anweisungen dann ausgegeben, wenn der ULANG-Profilparameter auf 1 gesetzt ist und die Map ohne die Anweisungen, wenn er auf 2 gesetzt ist.

119 Dialog-Gestaltung

- Feldabhängige Verarbeitung ... 864
- Einfachere Programmierung ... 866
- Zeilenabhängige Verarbeitung ... 867
- Spaltenabhängige Verarbeitung ... 868
- Verarbeitung aufgrund von Funktionstasten ... 869
- Verarbeitung aufgrund der Namen von Funktionstasten ... 870
- Verarbeitung von Daten außerhalb des aktiven Fensters ... 870
- Daten vom Bildschirm kopieren ... 874
- Statements REINPUT/REINPUT FULL ... 877
- Objektorientierte Datenverarbeitung — der Natural-Kommando-Prozessor ... 878

Dieses Kapitel beschreibt, wie zeichenbasierte Benutzeroberflächen erstellt werden können, die einen einfachen und flexiblen Benutzerdialog ermöglichen.

Anmerkung: Informationen zur Gestaltung von grafischen Benutzeroberflächen (GUI) finden Sie unter *Introduction to Event-Driven Programming*.

Feldabhängige Verarbeitung

Feldabhängige Verarbeitung — *CURS-FIELD und POS(fieldname)

Mit der Systemvariablen `*CURS-FIELD` zusammen mit der Systemfunktion `POS(`*`field-name`*`)` können Sie eine Verarbeitung davon abhängig machen, in welchem Feld der Cursor sich gerade befindet, wenn der Benutzer EINGABE drückt.

`*CURS-FIELD` enthält die interne Identifikation des Feldes, in dem der Cursor sich gerade befindet; diese Systemvariable kann nicht allein, sondern nur zusammen mit `POS(`*`field-name`*`)` benutzt werden.

Mit `*CURS-FIELD` und `POS(`*`field-name`*`)` können Sie z.B. dem Benutzer die Möglichkeit geben, eine Funktion auszuwählen, indem er einfach den Cursor auf ein bestimmtes Feld platziert und EINGABE drückt.

Das Beispiel unten demonstriert eine solche Anwendung:

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-FIELD = POS(#EMP) OR #EMP = 'X' OR #CODE = 1
    FETCH 'LISTEMP'
  WHEN *CURS-FIELD = POS(#CAR) OR #CAR = 'X' OR #CODE = 2
    FETCH 'LISTCAR'
   WHEN NONE
    REINPUT 'PLEASE MAKE A VALID SELECTION'
END-DECIDE
END
```

Und das Ergebnis:

```
                                SAMPLE MAP

                  Please select a function


                  1.) Employee information  _ <== Cursor auf Feld
                  2.) Vehicle information   _


                      Enter code:  _

To select a function, do one of the following:

- place the cursor on the input field next to desired function and press ENTER
- mark the input field next to desired function with an X and press ENTER
- enter the desired function code (1 or 2) in the 'Enter code' field and press
  ENTER
```

Wenn der Benutzer den Cursor auf das Eingabefeld (#EMP) neben `Employee information` platziert und EINGABE drückt, gibt das Programm `LISTEMP` eine Liste der Mitarbeiter aus:

```
Page      1                                                2001-01-22  09:39:32

        NAME
--------------------

ABELLAN
ACHIESON
ADAM
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
AECKERLE
AFANASSIEV
AFANASSIEV
AHL
AKROYD
```

Anmerkungen:

1. In Natural for Ajax-Anwendungen dient `*CURS-FIELD` zur Identifikation des Operanden, welcher den Wert des Control darstellt, welches den Eingabefokus hat. Sie können `*CURS-FIELD` in

Verbindung mit der `POS`-Funktion benutzen, um eine Prüfung auf das Control, das den Eingabefokus hat, zu veranlassen und die Verarbeitung in Abhängigkeit von diesem Zustand durchzuführen.

2. Die Werte von `*CURS-FIELD` und `POS(`*`field-name`*`)` dienen nur der internen Identifikation der Felder. Sie können für arithmetische Operationen nicht verwendet werden.

Einfachere Programmierung

Systemfunktion POS

Die Natural-Systemfunktion `POS(`*`field-name`*`)` enthält die interne Identifikation des Feldes, dessen Name mit der Systemfunktion angegeben wird.

`POS(`*`field-name`*`)` identifiziert ein bestimmtes Feld, unabhängig von seiner Position in einer Map. Auch wenn sich die Reihenfolge und Anzahl der Felder in einer Map ändert, identifiziert `POS(`*`field-name`*`)` nach wie vor eindeutig dasselbe Feld. Damit genügt zum Beispiel ein einziges `REINPUT`-Statement, um es von der Programmlogik abhängig zu machen, welches Feld `MARK`iert werden soll.

Anmerkung: Der Wert von `POS(`*`field-name`*`)` dient nur zur internen Identifikation der Felder. Er kann nicht für arithmetische Operationen benutzt werden.

Beispiel:

```
...
DECIDE ON FIRST VALUE OF ...
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD1)
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD2)
   ...
END-DECIDE
...
REINPUT ... MARK #FIELDX
...
```

Weitere Einzelheiten zu `*CURS-FIELD` und `POS(`*`field-name`*`)` siehe *Systemvariablen* und *Systemfunktionen*.

Zeilenabhängige Verarbeitung

Systemvariable *CURS-LINE

Mit der Systemvariablen *CURS-LINE können Sie eine Verarbeitung davon abhängig machen, in welcher Zeile der Cursor gerade steht, wenn der Benutzer EINGABE drückt.

Mit dieser Variablen können Sie z.B. benutzerfreundliche Menüs gestalten. Bei entsprechender Programmierung braucht der Benutzer lediglich den Cursor in die Zeile der gewünschten Menü-Funktion zu platzieren und EINGABE zu drücken, um die Funktion auszuführen.

Die Cursor-Position bezieht sich auf das aktive Fenster, unabhängig von der Position auf dem physischen Bildschirm.

Anmerkung: Die Meldungszeile, Funktionstastenleiste und Statistikzeile/Infoline zählen nicht als Datenzeilen auf dem Bildschirm.

Das Beispiel unten veranschaulicht die Möglichkeiten einer zeilenabhängigen Verarbeitung, die die Systemvariable *CURS-LINE bietet. Wenn der Benutzer in der Map EINGABE drückt, prüft das Programm, ob der Cursor in Zeile 8 des Bildschirms steht. Diese Zeile enthält die Option Employee information. Trifft dies zu, wird das Programm LISTEMP ausgeführt, das eine Liste der Mitarbeiter ausgibt.

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-LINE = 8
     FETCH 'LISTEMP'
   WHEN NONE
     REINPUT 'PLACE CURSOR ON LINE OF OPTION YOU WISH TO SELECT'
END-DECIDE
END
```

Ausgabe:

```
                                Company Information


                    Please select a function


                 [] 1.) Employee information

                    2.) Vehicle information









    Place the cursor on the line of the option you wish to select and press
    ENTER
```

Der Benutzer platziert den durch **[ ]** dargestellten Cursor in die Zeile der gewünschten Option und drückt EINGABE: das entsprechende Programm wird ausgeführt.

Spaltenabhängige Verarbeitung

Systemvariable *CURS-COL

Die Systemvariable `*CURS-COL` wird analog zu der Systemvariablen `*CURS-LINE` (wie oben beschrieben) benutzt. Mit `*CURS-COL` können Sie eine Verarbeitung davon abhängig machen, in welcher Spalte der Cursor steht.

Verarbeitung aufgrund von Funktionstasten

Systemvariable *PF-KEY

Oft ist eine funktionstastenabhängige Verarbeitung erwünscht.

Diese wird mittels des `SET KEY`-Statements und der Systemvariablen `*PF-KEY` und einer Änderung der Standard-Einstellungen der Map (`Standard Keys` = `Y`) erreicht.

Das `SET KEY`-Statement weist während der Ausführung des Programms Funktionstasten Funktionen zu. Die Systemvariable `*PF-KEY` enthält die Identifikation der vom Benutzer zuletzt gedrückten Funktionstaste.

Das nachstehende Beispiel veranschaulicht die Anwendung von `SET KEY` mit `*PF-KEY`.

```
...
SET KEY PF1
*
INPUT USING MAP 'DEMO&'
IF *PF-KEY = 'PF1'
  WRITE 'Help is currently not active'
END-IF
...
```

Das `SET KEY`-Statement aktiviert PF1 als Funktionstaste.

Das `IF`-Statement bestimmt, welche Aktionen erfolgen sollen, wenn der Benutzer PF1 drückt.

Beim Programmablauf wird der aktuelle Inhalt der Systemvariablen `*PF-KEY` geprüft; wenn sie PF1 enthält, wird die entsprechende Aktion ausgeführt.

Weitere Einzelheiten zum Statement `SET KEY` und der Systemvariable `*PF-KEY` finden Sie in der *Statements*- bzw. *Systemvariablen*-Dokumentation.

Verarbeitung aufgrund der Namen von Funktionstasten

Systemvariable *PF-NAME

Oft wird eine bestimmte Verarbeitung durch Drücken einer Funktionstaste ausgelöst. Noch mehr Komfort wird durch die Systemvariable *PF-NAME geboten. Mit ihr können Sie eine Verarbeitung von dem Namen einer Funktion abhängig machen, anstatt von einer Funktion.

Die Systemvariable *PF-NAME enthält den Namen der zuletzt gedrückten Funktionstaste (d.h. den Namen, der der Funktionstaste mit der NAMED-Klausel des SET KEY-Statements zugewiesen wurde).

Wünschen Sie z.B., dass der Benutzer die Hilfe-Funktion durch Drücken von wahlweise PF3 oder PF12 aufrufen kann, weisen Sie beiden Tasten den gleichen Namen (im folgenden Beispiel: INFO) zu. Drückt der Benutzer eine dieser beiden Funktionstasten, wird die im IF-Statement definierte Verarbeitung ausgelöst.

```
...
SET KEY PF3  NAMED 'INFO'
        PF12 NAMED 'INFO'
INPUT USING MAP 'DEMO&'
IF *PF-NAME = 'INFO'
  WRITE 'Help is currently not active'
END-IF
...
```

Die mit NAMED definierten Funktionsnamen erscheinen in der Funktionstastenleiste:

Verarbeitung von Daten außerhalb des aktiven Fensters

Die folgenden Themen werden behandelt:

- Systemvariable *COM
- Beispiel für die Benutzung von *COM

- Positionierung des Cursors auf *COM — Terminalkommando %T*

Systemvariable *COM

Wie im Abschnitt *Bildschirm-Gestaltung* — ***Fenster*** weiter oben beschrieben, ist jeweils nur *ein* Fenster aktiv. In der Regel bedeutet dies, dass Eingaben nur innerhalb dieses Fensters möglich sind.

Mit der Systemvariablen `*COM`, die als Kommunikationsbereich betrachtet werden kann, ist es möglich, Eingaben auch außerhalb des aktiven Fensters zu machen.

Die Voraussetzung hierfür ist, dass die Map `*COM` als modifizierbares Feld enthält. Der Benutzer kann dann in dieses Feld Daten eingeben, auch wenn ein Fenster aktiv ist. Eine weitere Verarbeitung kann vom Inhalt der `*COM`-Variablen abhängig gemacht werden.

Auf diese Weise können Sie Benutzeroberflächen einrichten, wie sie z.B. bei Con-nect, dem Büro-Kommunikationssystem der Software AG implementiert sind: hier hat der Benutzer immer die Möglichkeit, Daten in die Kommandozeile einzugeben, auch wenn ein Fenster mit eigenen Eingabefeldern aktiv ist.

Beachten Sie, dass der Inhalt von `*COM` nur dann gelöscht wird, wenn die Natural-Session beendet wird.

Beispiel für die Benutzung von *COM

Im folgenden Beispiel führt das Programm `ADD` eine einfache Addition der Daten, die in der Map eingegeben werden, durch. In dieser Map wurde `*COM` als modifizierbares Feld definiert (am unteren Rand der Map), und zwar mit der im `AL`-Feld des `Extended Field Editing` angegebenen Länge. Das Ergebnis der Berechnung wird in einem Fenster ausgegeben. Obwohl dieses Fenster keine Eingabefelder hat, kann der Benutzer dennoch Eingaben machen, und zwar in das `*COM`-Feld außerhalb des Fensters.

Programm ADD:

```
DEFINE DATA LOCAL
1 #VALUE1 (N4)
1 #VALUE2 (N4)
1 #SUM3 (N8)
END-DEFINE
*
DEFINE WINDOW EMP
  SIZE 8*17
  BASE 10/2
  TITLE 'Total of Add'
  CONTROL SCREEN
  FRAMED POSITION SYMBOL BOT LEFT
*
```

```
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
*
IF *COM = 'M'
  FETCH 'MULTIPLY' #VALUE1 #VALUE2
END-IF
END
```

Ausgabe des Programms ADD:

```
                   Map to Demonstrate Windows with *COM

                                    CALCULATOR

                        Enter values you wish to calculate


                        Value 1:  12__
                        Value 2:  12__
   +-Total of Add-+
   !              !
   ! Value 1 +    !
   ! Value 2 =    !
   !              !
   !         24   !
   !              !
   +--------------+


    Next line is input field (*COM) for input outside the window:
```

Durch Eingabe von M wird die Funktion Multiplikation angestoßen; die beiden Werte aus der Eingabemaske werden miteinander multipliziert und das Ergebnis in einem zweiten Fenster ausgegeben:

```
                   Map to Demonstrate Windows with *COM

                                CALCULATOR

                        Enter values you wish to calculate


                        Value 1:  12__
                        Value 2:  12__
  +-Total of Add-+                            +--------------+
  !              !                            !              !
  ! Value 1 +    !                            ! Value 1 x    !
  ! Value 2 =    !                            ! Value 2 =    !
  !              !                            !              !
  !         24   !                            !      144     !
  !              !                            !              !
  +--------------+                            +--------------+


   Next line is input field (*COM) for input outside the window:
               M
```

Positionierung des Cursors auf *COM — Terminalkommando %T*

Wenn ein Fenster aktiv ist und keine Eingabefelder (`AD=M` oder `AD=A`) enthält, wird der Cursor standardmäßig in die obere linke Ecke des Fensters positioniert.

Mit dem Terminalkommando `%T*` können Sie den Cursor auf die Systemvariable `*COM` außerhalb des Fensters positionieren, wenn das aktive Fenster keine Eingabefelder enthält.

Bei erneuter Eingabe von `%T*` wird die standardmäßige Positionierung des Cursors wieder aktiv.

Beispiel:

```
...
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET CONTROL 'T*'
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
...
```

Daten vom Bildschirm kopieren

Folgende Themen werden behandelt:

- Terminalkommandos %CS und %CC
- Eine Ausgabezeile eines Reports zur weiteren Verarbeitung auswählen

Terminalkommandos %CS und %CC

Mit diesen Terminalkommandos können sie Teile eines Bildschirms in den Natural-Stack (%CS) bzw. in die Systemvariable *COM (%CC) kopieren. Die geschützten Daten einer bestimmten Bildschirmzeile werden Feld für Feld kopiert.

Eine vollständige Beschreibung dieser Terminalkommandos finden Sie in der *Terminalkommandos*-Dokumentation.

Befinden sich die Daten im Stack oder in *COM, stehen sie zur weiteren Verarbeitung zur Verfügung. Mit diesen Kommandos können sie benutzerfreundliche Anwendungen wie im nachfolgenden Beispiel erstellen.

Eine Ausgabezeile eines Reports zur weiteren Verarbeitung auswählen

Im folgenden Beispiel gibt das Programm COM1 eine Liste der Mitarbeiter von Abellan bis Alestia aus.

Programm COM1:

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
END-DEFINE
*
READ EMP BY NAME STARTING FROM 'ABELLAN' THRU 'ALESTIA'
  DISPLAY NAME
END-READ
FETCH 'COM2'
END
```

Ausgabe des Programms `COM1`:

```
Page      1                                               2006-08-12  09:41:21

        NAME
--------------------

ABELLAN
ACHIESON
ADAM
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
AECKERLE
AFANASSIEV
AFANASSIEV
AHL
AKROYD
ALEMAN
ALESTIA
MORE
```

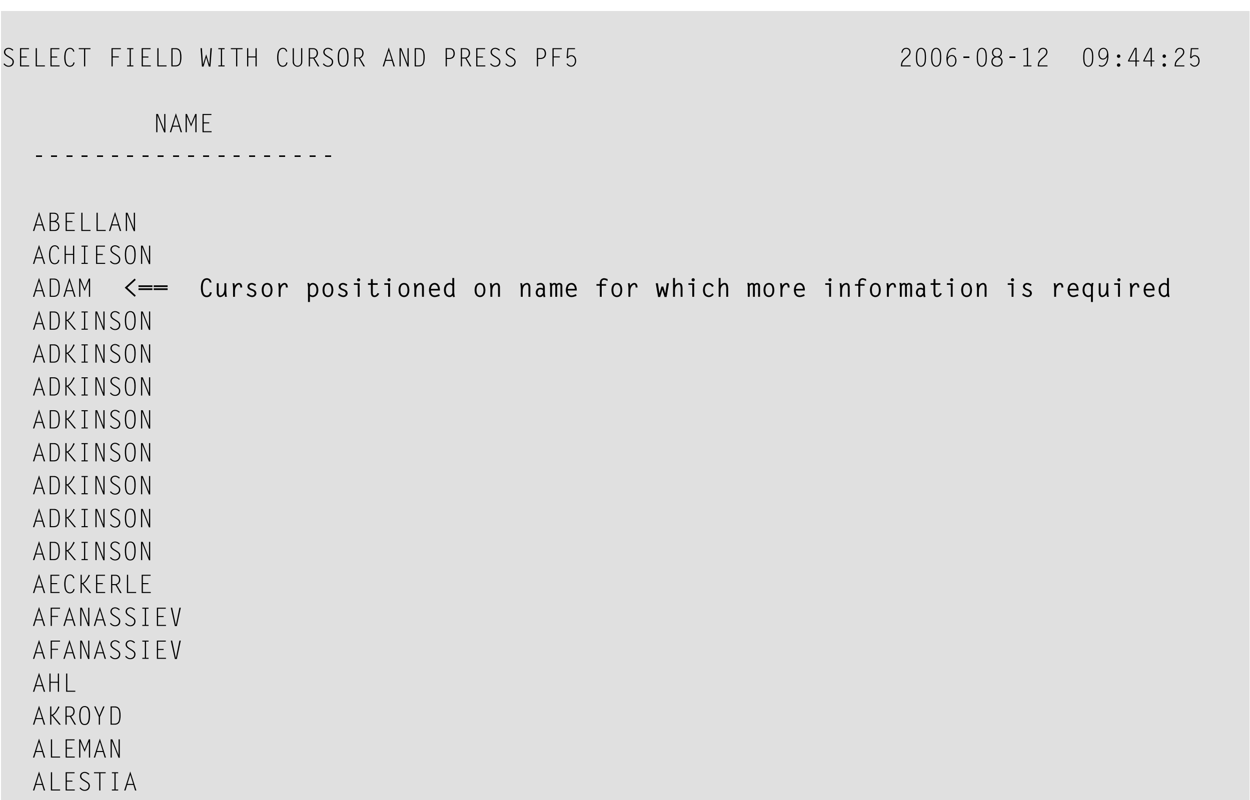
Die Kontrolle wird nun an das Programm `COM2` übergeben.

Programm COM2:

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
1 SELECTNAME (A20)
END-DEFINE
*
SET KEY PF5 = '%CCC'
*
INPUT NO ERASE 'SELECT FIELD WITH CURSOR AND PRESS PF5'
*
MOVE *COM TO SELECTNAME
FIND EMP WITH NAME = SELECTNAME
  DISPLAY NAME PERSONNEL-ID
END-FIND
END
```

In diesem Programm ist das Terminalkommando `%CCC` der Funktionstaste PF5 zugeordnet. Das Terminalkommando kopiert alle geschützten Daten von der Zeile, in der sich der Cursor befindet, in die Systemvariable `*COM`. Diese Daten stehen dann zur weiteren Verarbeitung zur Verfügung. Die Art der Verarbeitung wird in den fett hervorgehobenen Zeilen des Beispielprogramms festgelegt.

Jetzt platziert der Benutzer den Cursor auf den Namen, der ihn interessiert, drückt PF5, und weitere Daten dieses Mitarbeiters werden ausgegeben.



```
SELECT FIELD WITH CURSOR AND PRESS PF5                     2006-08-12  09:44:25

          NAME
  --------------------

  ABELLAN
  ACHIESON
  ADAM  <==  Cursor positioned on name for which more information is required
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  AECKERLE
  AFANASSIEV
  AFANASSIEV
  AHL
  AKROYD
  ALEMAN
  ALESTIA
```

In diesem Fall wird die Personalnummer (`PERSONNEL ID`) des ausgewählten Mitarbeiters angezeigt:

```
  Page      1                                              2006-08-12  09:44:52

          NAME         PERSONNEL
                          ID
  -------------------- ---------

  ADAM                 50005800
```

Statements REINPUT/REINPUT FULL

Das Statement `REINPUT` dient dazu, zu einem `INPUT`-Statement zurückzukehren und dieses erneut auszuführen. In der Regel wird es dazu benutzt, eine Fehlermeldung auszugeben, die dem Benutzer sagt, dass auf das `INPUT`-Statement hin ungültige Daten eingegeben wurden.

Wenn Sie die Option `FULL` in einem `REINPUT`-Statement angeben, wird das entsprechende `INPUT`-Statement vollständig neu ausgeführt:

- Bei einem normalen `REINPUT`-Statement (ohne `FULL`-Option) werden Inhalte von Variablen, die zwischen `INPUT`- und `REINPUT`-Statement geändert wurden, nicht angezeigt; d.h. alle Variablen auf dem Schirm zeigen den Inhalt, den Sie hatten, als das `INPUT`-Statement ursprünglich ausgeführt wurde.
- Bei einem `REINPUT FULL`-Statement werden alle nach der ersten Ausführung des `INPUT`-Statements gemachten Änderungen sichtbar, wenn das `INPUT`-Statement erneut ausgeführt wird; d.h. alle Variablen auf dem Schirm haben den Inhalt, den sie zum Zeitpunkt der Ausführung des `REINPUT`-Statements hatten.
- Wollen Sie zusätzlich den Cursor auf ein bestimmtes Feld positionieren, so können Sie die `MARK`-Option verwenden. Um auf eine bestimmte Position innerhalb eines Feldes zu positionieren, geben Sie die `MARK POSITION`-Option an.

Das Beispiel unten veranschaulicht die Anwendung von `REINPUT FULL` mit `MARK POSITION`.

```
DEFINE DATA LOCAL
1 #A (A10)
1 #B (N4)
1 #C (N4)
END-DEFINE
*
INPUT (AD=M) #A #B #C
IF #A = ' '
  COMPUTE #B = #B + #C
  RESET #C
  REINPUT FULL 'Enter a value' MARK POSITION 5 IN *#A
END-IF
END
```

Der Benutzer gibt 3 in das Feld #B und 3 in das Feld #C ein und drückt EINGABE.

```
  #A              #B      3 #C      3
```

Das Programm verlangt, dass das Feld #A einen Wert enthält. Das Statement REINPUT FULL mit MARK POSITION 5 IN *#A gibt den Eingabeschirm erneut aus, und zwar mit der geänderten Variable #B, die jetzt den Wert 6 enthält (Stand nach der COMPUTE-Berechnung). Der Cursor wird an die 5. Stelle in das für Neueingaben bereite Feld #A platziert.

```
Enter name of field
  #A    _         #B      6 #C      0



Enter a value
```

Das gleiche Statement ohne die FULL-Option würde folgenden Bildschirm ausgeben. Beachten Sie, dass die Variablen #B and #C auf den Stand zur Ausführungszeit des INPUT-Statements zurückgesetzt wurden (beide Felder enthalten den Wert 3).

```
  #A    _         #B      3 #C      3
```

Objektorientierte Datenverarbeitung — der Natural-Kommando-Prozessor

Der Natural-Kommando-Prozessor wird zur Definition und Steuerung der Navigation innerhalb einer Anwendung benutzt.

- Der *Entwicklungsteil* ist die Utility SYSNCP. Mit dieser Utility definieren Sie Kommandos, d.h. Kombinationen von Schlüsselwörtern, und die Aktionen, die als Reaktion auf die Ausführung dieser Kommandos ausgeführt werden sollen. Aus Ihren Definitionen erzeugt SYSNCP Entscheidungstabellen, die festlegen, was passiert, wenn ein Benutzer ein Kommando eingibt.
- Der *Laufzeitteil* ist das Statement PROCESS COMMAND. Dieses Statement wird benutzt, um den Kommando-Prozessor in einem Natural-Programm aufzurufen. Im Statement geben Sie den Namen der SYSNCP-Tabelle an, die benutzt wird, um die Dateneingabe eines Benutzers zu diesem Zeitpunkt zu verarbeiten.

Weitere Informationen zum Natural-Kommando-Prozessor siehe *SYSNCP Utility* in der *Utilities*-Dokumentation und das Statement PROCESS COMMAND in der *Statements*-Dokumentation.

120 Natural Native Interface

- Introduction
- Interface Library and Location
- Interface Versioning
- Interface Access
- Interface Instances and Natural Sessions
- Interface Functions
- Parameter Description Structure
- Natural Data Types
- Flags
- Return Codes
- Natural Exception Structure
- Interface Usage
- Threading Issues

121 Introduction

The Natural Native Interface enables an application to execute Natural code in its own process context through function calls according to the C calling convention. The interface consists of a DLL that contains a set of interface functions. These functions include initialization and uninitialization of a Natural session, logging on to a specific Natural library and execution of individual Natural modules. The calling application loads the interface library dynamically with operating system calls and then locates and calls the interface functions.

An example C program *nnisample.c* that shows the usage of the interface is contained in *%NATDIR%\%NATVERS%\samples\sysexnni*.

The Natural modules called by the C program *nnisample.c* are contained in the Natural library SYSEXNNI.

122 Interface Library and Location

The interface consists of a DLL that exports a set of functions. The individual functions are described in ***Interface Functions***. The interface DLL is called *natni.dll* and is contained in the Natural *bin* directory.

When executing a program that uses the Natural Native Interface, the Natural *bin* directory must be defined in the environment variable PATH, so that the calling program can locate the interface library and all dependent libraries.

123 Interface Versioning

The Natural Native Interface might change in future versions of Natural. Natural versions that provide a modified interface will support previous interface versions in parallel, until a point in time that is determined by Software AG and is announced in time. To access an instance of a specific version of the interface, the application calls the function `nni_get_interface`. The application passes the required interface version number to the function and receives a structure with function pointers in return. The application may also request the most recent interface version, without specifying the interface version explicitly.

124 Interface Access

In order to access the interface, an application loads the interface library using a platform dependent system call.

Then the application locates the address of the function `nni_get_interface`, again using a platform dependent system call. Once the application has located the central function `nni_get_interface`, it requests an instance of the interface by calling the function `nni_get_interface` and specifying the desired interface version. The resulting structure contains the interface function pointers.

After having finished using the interface functions, the application unloads the interface library using a platform dependent system call.

The sample program *nnisample.c* demonstrates the interface. Also the platform dependent mechanism of loading the interface library and the access to the function `nni_get_interface` is illustrated by this sample program.

125 Interface Instances and Natural Sessions

The function `nni_get_interface` returns a pointer to an instance of the Natural Native Interface. One interface instance can host one Natural session at a time. An application initializes a Natural session by calling the function `nni_initialize` on a given interface instance. It uninitializes the Natural session by calling `nni_uninitialize` on that interface instance. After that it can initialize a new Natural session on the same interface instance.

It is implementation dependent if multiple interface instances and thus multiple Natural sessions can be maintained per process or per thread. In the current implementation of Natural on Windows and UNIX one process can host one Natural session at a time. Consequently, every call to `nni_get_interface` in one process yields the same interface instance. However, this unique interface instance can be used alternating by several concurrently running threads. The thread synchronization is implicitly performed by the interface functions themselves. Optionally it can be performed by the application explicitly. The interface provides the required synchronization functions `nni_enter`, `nni_try_enter` and `nni_leave`.

126 Interface Functions

- nni_get_interface 893
- nni_free_interface 894
- nni_initialize 894
- nni_is_initialized 896
- nni_uninitialize 897
- nni_enter 897
- nni_try_enter 898
- nni_leave 898
- nni_logon 899
- nni_logoff 900
- nni_callnat 901
- nni_function 902
- nni_create_object 903
- nni_send_method 904
- nni_get_property 906
- nni_set_property 907
- nni_delete_object 909
- nni_create_parm 910
- nni_create_module_parm 911
- nni_create_method_parm 912
- nni_create_prop_parm 913
- nni_parm_count 914
- nni_init_parm_s 915
- nni_init_parm_sa 916
- nni_init_parm_d 917
- nni_init_parm_da 918
- nni_get_parm_info 919
- nni_get_parm 920
- nni_get_parm_array 921
- nni_get_parm_array_length 922
- nni_put_parm 923
- nni_put_parm_array 924

- nni_resize_parm_array 925
- nni_delete_parm 926
- nni_from_string 927
- nni_to_string 928

nni_get_interface

Syntax

```
int nni_get_interface( int iVersion, void** ppnni_func );
```

The function returns an instance of the Natural Native Interface.

An application calls this function after having retrieved and loaded the interface library with platform depending system calls. The function returns a pointer to a structure that contains function pointers to the individual interface functions. The functions returned in the structure may differ between interface versions.

Instead of a specific interface version, the caller can also specify the constant `NNI_VERSION_CURR`, which always refers to the most recent interface version. The interface version number belonging to a given Natural version is defined in the header file *natni.h* that is delivered with that version. In Natural Version *n.n*, the interface version number is defined as `NNI_VERSION_`*nn*. `NNI_VERSION_CURR` is also defined as `NNI_VERSION_`*nn*. If the Natural version against which the function is called does not support the requested interface version, the error code `NNI_RC VERSION_ERROR` is returned. Otherwise the return code is `NNI_RC_OK`.

The pointer returned by the function represents one instance of the interface. In order to use this interface instance, the application holds on to that pointer and passes it to subsequent interface calls.

Usually the application will subsequently initialize a Natural session by calling `nni_initialize` on the given instance. After the application has finished using that Natural session, it calls `nni_uninitialize` on that instance. After that it can initialize a different Natural session on the same interface instance. After the application has finished using the interface instance entirely, it calls `nni_free_interface` on that instance.

Parameters

Parameter	Meaning
`iVersion`	Interface version number. (`NNI_VERSION_`*nn* or `NNI_VERSION_CURR`).
`ppnni_func`	Points to an NNI interface instance on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_PARM_ERROR	
NNI_RC_VERSION_ERROR	

nni_free_interface

Syntax

```
int nni_free_interface(void* pnni_func);
```

An application calls this function after it has finished using the interface instance and has uninitialized the Natural session it hosts. The function frees the resources occupied by that interface instance.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_initialize

Syntax

```
int nni_initialize(void* pnni_func, const char* szCmdLine, void*, void*);
```

The function initializes a Natural session with a given command line. The syntax and semantics of the command line is the same as when Natural is started interactively. If a Natural session has

already been initialized on the given interface instance, that session is implicitly uninitialized before the new session is initialized.

The command line must be specified in the way that the Natural initialization can be completed without user interaction. This means especially that if a program is passed on the stack or a startup program is specified, that program must not perform an `INPUT` statement that is not satisfied from the stack. Otherwise the subsequent behavior of the Natural session is undetermined.

The Natural session is initialized as batch session and in server mode. This means that the usage of certain statements and commands in the executed Natural modules is restricted. These restrictions and error conditions are the same as documented in the section *Using Statements and Commands in a NaturalX Server Environment* of the *Operations* documentation.

When initalizing a Natural session under Natural Security, the command line must contain a `LOGON` command to a freely chosen default library under which the session will be started, and an appropriate user ID and password.

Example:

```
int iRes =
pnni_func->nni_initialize( pnni_func, "STACK=(LOGON,MYLIB,MYUSER,MYPASS)", 0, 0);
```

If the application later calls `nni_logon` to a different library with a different user ID and afterwards calls `nni_logoff`, the Natural session will be reset to the library and user ID that was passed during `nni_initialize`.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szCmdLine`	Natural command line. May be a null pointer.
`void*`	For future use. Must be a null pointer.
`void*`	For future use. Must be a null pointer.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_PARM_ERROR	
rc, where *rc* < NNI_RC_SERR_OFFSET	Natural startup error. The real Natural startup error number as documented in *Natural Startup Errors* (which is part of the *Operations* documentation) can be determined by the following calculation: *startup-error-nr* = - (*rc* - NNI_RC_SERR_OFFSET) Warnings that occur during session initialization are ignored.
> 0	Natural error number.

nni_is_initialized

Syntax

```
int nni_is_initialized( void* pnni_func, int* piIsInit );
```

The function checks if the interface instance contains an initialized Natural session.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
piIsInit	Returns 0, if no Natural session is initialized, a non-zero value otherwise.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_PARM_ERROR	

nni_uninitialize

Syntax

```
int nni_uninitialize(void* pnni_func);
```

The function uninitializes the Natural session hosted by the given interface instance.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	

nni_enter

Syntax

```
int nni_enter(void* pnni_func);
```

The function lets the current thread wait for exclusive access to the interface instance and the Natural session it hosts. A thread calls this function if it wants to issue a series of interface calls that may not be interrupted by other threads. The thread releases the eclusive access to the interface instance by calling nni_leave.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_try_enter

Syntax

```
int nni_try_enter(void* pnni_func);
```

The function behaves like `nni_enter` except that it does not block the thread and instead always returns immediately. If a different thread already has exclusive access to the interface instance, the function returns `NNI_RC_LOCKED`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_LOCKED	

nni_leave

Syntax

```
int nni_leave(void* pnni_func);
```

The function releases exclusive access to the interface instance and allows other threads to access that instance and the Natural session it hosts.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_logon

Syntax

```
int nni_logon(void* pnni_func, const char* szLibrary, const char* szUser, const
char* szPassword);
```

The function performs a LOGON to the specified Natural library.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szLibrary	Name of the Natural library.
szUser	Name of the Natural user. May be a null pointer, if the Natural session is not running under Natural Security or if AUTO=ON was used during initialization.
szPassword	Password of that user. May be a null pointer, if the Natural session is not running under Natural Security or if AUTO=ON was used during initialization..

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_logoff

Syntax

```
int nni_logoff(void* pnni_func);
```

The function performs a LOGOFF from the current Natural library. This corresponds to a LOGON to the previously active library and user ID.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_callnat

Syntax

```
int nni_callnat(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

The function calls a Natural subprogram.

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use one the functions `create_parm` or `create_module_parm` to create an appropriate parameter set for the subprogram.
- If you have used `create_parm`, use the functions `init_parm_*` to initialize each parameter to the appropriate Natural data format. If you have used `create_module_parm`, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter, using one the functions `nni_put_parm` or `nni_put_parm_array`.
- Call `nni_get_parm` on each parameter in the set. This fills the `parameter_description` structures.
- Pass the array of `parameter_description` structures to the function `nni_callnat`.
- After the call has been executed, extract the modified parameter values from the parameter set using the function `nni_get_parm` or `nni_get_parm_array`.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the Natural subprogram.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`.
`rgDesc`	An array of `parm_description` structures containing the parameters for the subprogram. If the subprogram does not expect parameters, the caller passes a null pointer.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during execution of the subprogram,this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended error information is returned.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_function

Syntax

```
int nni_callnat(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

The function calls a Natural subprogram.

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use one the functions `create_parm` or `create_module_parm` to create an appropriate parameter set for the subprogram.
- If you have used `create_parm`, use the functions `init_parm_*` to initialize each parameter to the appropriate Natural data format. If you have used `create_module_parm`, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter, using one the functions `nni_put_parm` or `nni_put_parm_array`.
- Call `nni_get_parm` on each parameter in the set. This fills the `parameter_description` structures.
- Pass the array of `parameter_description` structures to the function `nni_callnat`.
- After the call has been executed, extract the modified parameter values from the parameter set using the function `nni_get_parm` or `nni_get_parm_array`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szName	Name of the Natural subprogram.
iParm	Number of parameters. Indicates the number of occurrences of the array rgDesc.
rgDesc	An array of parm_description structures containing the parameters for the subprogram. If the subprogram does not expect parameters, the caller passes a null pointer.
pExcep	Pointer to a natural_exception structure. If a Natural error occurs during execution of the subprogram,this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended error information is returned.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_create_object

Syntax

```
int nni_create_object(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Creates a Natural object (an instance of a Natural class).

The function receives its parameters as a one-element array of parameter_description structures. The caller creates the structures using NNI functions in the following way:

- Use the function nni_create_parm to create parameter set with one element.
- Use the function nni_init_parm_s to initialize the parameter with the type HANDLE OF OBJECT.
- Call nni_get_parm_info on this parameter. This fills the parameter_description structure.
- Pass the parameter_description structure to the function nni_create_object.

- After the call has been executed, extract the modified parameter value from the parameter set using one the function `nni_get_parm`.

The parameters passed in `rgDesc` have the following meaning:

- The first (and only) parameter must be initialized with the data type `HANDLE OF OBJECT` and contains on return the Natural object handle of the newly created object.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the class.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`.
`rgDesc`	An array of `parm_description` structures containing the parameters for the object creation. The caller always passes one parameter, which will contain the object handle on return.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during object creation, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
`NNI_RC_NO_MEMORY`	
> 0	Natural error number.

nni_send_method

Syntax

```
int nni_send_method(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Sends a method call to a Natural object (an instance of a Natural class).

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use the function nni_create_parm or nni_create_method_parm to create a matching parameter set.
- If you have used create_parm, use the functions init_parm_* to initialize each parameter to the appropriate Natural data format. If you have used nni_create_method_parm, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter using one the functions nni_put_parm or nni_put_parm_array.
- Call nni_get_parm_info on each parameter in the set. This fills the parameter_description structures.
- Pass the array of parameter_description structures to the function nni_send_method.
- After the call has been executed, extract the modified parameter values from the parameter set using one of the nni_get_parm functions.

The parameters passed in rgDesc have the following meaning:

- The first parameter contains the object handle.
- The second parameter must be initialized to the data type of the method return value. If the method does not have a return value, the second parameter remains not initialized. On return from the method call, this parameter contains the return value of the method.
- The remaining parameters are the method parameters.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szName	Name of the method.
iParm	Number of parameters. Indicates the number of occurrences of the array rgDesc. This is always 2 + the number of method parameters.
rgDesc	An array of parm_description structures containing the parameters for the method. If the method does not expect parameters, the caller still passes two parameters, the first for the object handle and the second for the return value.
pExcep	Pointer to a natural_exception structure. If a Natural error occurs during execution of the method, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_get_property

Syntax

```
int nni_get_property(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Retrieves a property value of a Natural object (an instance of a Natural class).

The function receives its parameters as an array of parameter_description structures. The caller creates these structures using NNI functions in the following way:

- Use the function nni_create_parm or nni_create_method_parm to create a matching parameter set.
- If you have used create_parm, use the functions init_parm_* to initialize each parameter to the appropriate Natural data format. If you have used create_method_parm, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter using one the functions nni_put_parm or nni_put_parm_array.
- Call nni_get_parm_info on each parameter in the set. This fills the parameter_description structures.
- Pass the array of parameter_description structures to the function nni_send_method.
- After the call has been executed, extract the modified parameter values from the parameter set using one of the nni_get_parm functions.

The parameters passed in rgDesc have the following meaning:

- The first parameter contains the object handle.
- The second parameter is initialized to the data type of the property. On return from the property access, this parameter contains the property value.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the property.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`. This is always 2.
`rgDesc`	An array of `parm_description` structures containing the parameters for the property access. The caller always passes two parameters, the first for the object handle and the second for the returned property value.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during property access, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
`NNI_RC_NO_MEMORY`	
> 0	Natural error number.

nni_set_property

Syntax

```
int nni_set_property(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Assigns a property value to a Natural object (an instance of a Natural class).

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use the function `nni_create_parm` or `nni_create_prop_parm` to create a matching parameter set.
- If you have used `create_parm`, use the functions `init_parm_*` to initialize each parameter to the appropriate Natural data format. If you have used `create_prop_parm`, the parameters are

already initialized to the appropriate Natural data format. Assign a value to each parameter using one of the nni_put_parm functions.

- Assign a value to each parameter using one the functions nni_put_parm or nni_put_parm_array.
- Call nni_get_parm_info on each parameter in the set. This fills the parameter_description structures.
- Pass the array of parameter_description structures to the function nni_set_property.

The parameters passed in rgDesc have the following meaning:

- The first parameter contains the object handle.
- The second parameter contains the property value.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szName	Name of the property.
iParm	Number of parameters. Indicates the number of occurrences of the array rgDesc. This is always 2.
rgDesc	An array of parm_description structures containing the parameters for the property access. The caller always passes two parameters, the first for the object handle and the second for the property value.
pExcep	Pointer to a natural_exception structure. If a Natural error occurs during property access, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_delete_object

Syntax

```
int nni_delete_object(void* pnni_func, int iParm, struct parameter_description*
rgDesc, struct natural_exception* pExcep);
```

Deletes a Natural object (an instance of a Natural class) created with `nni_create_object`.

The function receives its parameters as a one-element array of `parameter_description` structures. The caller creates the structures using NNI functions in the following way:

- Use the function `nni_create_parm` to create parameter set with one element.
- Use the function `nni_init_parm_s` to initialize the parameter with the type `HANDLE OF OBJECT`.
- Assign a value to the parameter using one the functions `nni_put_parm`.
- Call `nni_get_parm_info` on this parameter. This fills the `parameter_description` structure.
- Pass the `parameter_description` structure to the function `nni_delete_object`.

The parameters passed in `rgDesc` have the following meaning:

- The first (and only) parameter must be initialized with the data type `HANDLE OF OBJECT` and contains the Natural object handle of the object to be deleted.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the class.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`. This is always 1.
`rgDesc`	An array of `parm_description` structures containing the parameters for the object creation. The caller always passes one parameter, which contains the object handle.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during object creation, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_create_parm

Syntax

```
int nni_create_parm(void* pnni_func, int iParm, void** pparmhandle);
```

Creates a set of parameters that can be passed to a Natural module.

The parameters contained in the set are not yet initialized to specific Natural data types. Before using the parameter set in a call to `nni_callnat`, `nni_create_object`, `nni_send_method`, `nni_set_property` or `nni_get_property`:

- Initialize each parameter to the required Natural data type using one of the functions `nni_init_parm_s`, `nni_init_parm_sa`, `nni_init_parm_d` or `nni_init_parm_da`.
- Assign a value to each parameter using one of the functions `nni_put_parm` or `nni_put_parm_array`.
- Turn each parameter into a `parm_description` structure using the function `nni_get_parm_info`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Requested number of parameters. The maxinum number of parameters is 32767.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
> 0	Natural error number.

nni_create_module_parm

Syntax

```
int nni_create_module_parm(void* pnni_func, char chType, const char* szName, void**
pparmhandle);
```

Creates a set of parameters that can be used in a call to `nni_callnat`. The function enables an application to dynamically explore the signature of a callable Natural module.

The parameters contained in the returned set are already initialized to Natural data types according to the parameter data area of the specified module. Before using the parameter set in a call to `nni_callnat`:

- Assign a value to each parameter using one of the functions `nni_put_parm` or `nni_put_parm_array`.
- Turn each parameter into a `parm_description` structure using the function `nni_get_parm_info`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
chType	Type of the Natural module. Always "N" (for subprogram).
szName	Name of the Natural module.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_create_method_parm

Syntax

```
int nni_create_method_parm( void* pnni_func, const char* szClass, const char*
szMethod, void** pparmhandle );
```

Creates a set of parameters that can be used in a call to nni_send_method. The function enables an application to dynamically explore the signature of a method of a Natural class.

The returned parameter set contains not only the method parameters, but also the other parameters required by nni_send_method. This means: If the method has n parameters, the parameter set contains $n + 2$ parameters.

- The first parameter in the set is initialized to the data type HANDLE OF OBJECT.
- The second parameter in the set is initialized to the data type of the method return value. If the method does not have a return value, the second parameter is not initialized.
- The remaining parameters in the set are initialized to the data types of the method parameters.

Before using the parameter set in a call to nni_send_method:

- Assign a value to each parameter using one of the functions nni_put_parm or nni_put_parm_array.
- Turn each parameter into a parm_description structure using the function nni_get_parm_info.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szClass	Name of the Natural class.
szMethod	Name of the Natural method.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_create_prop_parm

Syntax

```
int nni_create_prop_parm(void* pnni_func, const char* szClass, const char*
szProp,void** pparmhandle);
```

Creates a set of parameters that can be used in a call to `nni_get_property` or `nni_set_property`. The returned parameter set contains all parameters required by `nni_get_property` or `nni_set_property`. The function enables an application to determine the data type of a property of a Natural class.

- The first parameter in the set is initialized to the data type `HANDLE OF OBJECT`.
- The second parameter in the set is initialized to the data type of the property.

Before using the parameter set in a call to `nni_get_property` or `nni_set_property`:

- Assign a value to each parameter using one of the functions `nni_put_parm` or `nni_put_parm_array`.
- Turn each parameter into a `parm_description` structure using the function `nni_get_parm_info`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szClass	Name of the Natural class.
szProp	Name of the Natural property.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_parm_count

Syntax

```
int nni_parm_count( void* pnni_func, void* parmhandle, int* piParm )
```

The function retrieves the number of parameters in a parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
parmhandle	Pointer to a parameter set.
piParm	Returns the number of parameters in the parameter set.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	

nni_init_parm_s

Syntax

```
int nni_init_parm_s(void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iLength, int iPrecision, int iFlags);
```

Initializes a parameter in a parameter set to a static data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter.
iLength	Natural length of the parameter.
iPrecision	Number of decimal places (NNI_TYPE_NUM and NNI_TYPE_PACK only).
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_LENGTH	

nni_init_parm_sa

Syntax

```
int nni_init_parm_sa (void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iLength, int iPrecision, int iDim, int* rgiOcc, int iFlags);
```

Initializes a parameter in a parameter set to an array of a static data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter.
iLength	Natural length of the parameter.
iPrecision	Number of decimal places (NNI_TYPE_NUM and NNI_TYPE_PACK only).
iDim	Array dimension of the parameter.
rgiOcc	Three dimensional array of int values, indicating the occurrence count for each dimension. The occurrence count for unused dimensions must be specified as 0.
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED NNI_FLG_LBVAR_0 NNI_FLG_UBVAR_0 NNI_FLG_LBVAR_1 NNI_FLG_UBVAR_1

Parameter	Meaning
	NNI_FLG_LBVAR_2 NNI_FLG_UBVAR_2 If one of the NNI_FLG_*VAR* flags is set, the array is an x-array. In each dimension only the lower bound or the upper bound (not both) can be variable. Therefore for instance the flag IF4_FLG_LBVAR_0 may not be combined with IF4_FLG_UBVAR_0.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_LENGTH	
NNI_RC_BAD_DIM	
NNI_RC_BAD_BOUNDS	

nni_init_parm_d

Syntax

```
int nni_init_parm_d(void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iFlags);
```

Initializes a parameter in a parameter set to a dynamic data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter (NNI_TYPE_ALPHA or NNI_TYPE_BIN).
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	

nni_init_parm_da

Syntax

```
int nni_init_parm_da (void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iDim, int* rgiOcc, int iFlags);
```

Initializes a parameter in a parameter set to an array of a dynamic data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter (NNI_TYPE_ALPHA or NNI_TYPE_BIN).
iDim	Array dimension of the parameter.
rgiOcc	Three dimensional array of int values, indicating the occurrence count for each dimension. The occurrence count for unused dimensions must be specified as 0.
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED NNI_FLG_LBVAR_0 NNI_FLG_UBVAR_0 NNI_FLG_LBVAR_1 NNI_FLG_UBVAR_1 NNI_FLG_LBVAR_2 NNI_FLG_UBVAR_2

Parameter	Meaning
	If one of the NNI_FLG_*VAR* flags is set, the array is an x-array. In each dimension only the lower bound or the upper bound (not both) can be variable. Therefore for instance the flag IF4_FLG_LBVAR_0 may not be combined with IF4_FLG_UBVAR_0.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_DIM	
NNI_RC_BAD_BOUNDS	

nni_get_parm_info

Syntax

```
int nni_get_parm_info (void* pnni_func, int iParm, void* parmhandle, struct
parameter_description* pDesc);
```

Returns detailed information about a specific parameter in a parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
pDesc	Parameter description structure.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	

nni_get_parm

Syntax

```
int nni_get_parm(void* pnni_func, int iParm, void* parmhandle, int iBufferLength,
void* pBuffer);
```

Returns the value of a specific parameter in a parameter set. The value is returned in the buffer at the address specified in `pBuffer`, with the size specified in `iBufferLength`. On successful return, the buffer contains the data in Natural internal format. See ***Natural Data Types*** on how to interpret the contents of the buffer.

If the length of the parameter according to the Natural data type is greater than `iBufferLength`, Natural truncates the data to the given length and returns the code `NNI_RC_DATA_TRUNC`. The caller can use the function `nni_get_parm_info` to request the length of the parameter value in advance.

If the length of the parameter according to the Natural data type is smaller than `iBufferLength`, Natural fills the buffer according to the length of the parameter and returns the length of the copied data in the return code.

If the parameter is an array, the function returns the whole array in the buffer. This makes sense only for fixed size arrays of fixed size elements, because in other cases the caller cannot interpret the contents of the buffer. In order to retrieve an individual occurrence of an arbitrary array use the function `nni_get_parm_array`.

If no memory of the size specified in `iBufferLength` is allocated at the adress specified in `pBuffer`, the results of the operation are unpredictable. Natural only checks that `pBuffer` is not null.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
iBufferLength	Length of the buffer specified in pBuffer.
pBuffer	Buffer in which the value is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
= *n*, where *n* > 0	Successful operation, but only *n* bytes were returned in the buffer.

nni_get_parm_array

Syntax

```
int nni_get_parm_array(void* pnni_func, int parmnum, void* parmhandle, int
iBufferLength, void* pBuffer, int* rgiInd);
```

Returns the value of a specific occurrence of a specific array parameter in a parameter set. The only difference to nni_get_parm is that array indices can be specified. The indices for unused dimensions must be specified as 0.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
iBufferLength	Length of the buffer specified in pBuffer.
pBuffer	Buffer in which the value is returned.
rgiInd	Three dimensional array of int values, indicating a specific array occurrence. The indices start with 0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	
= *n*, where *n* > 0	Successful operation, but only *n* bytes were returned.

nni_get_parm_array_length

Syntax

```
int nni_get_parm_array_length(void* pnni_func, int iParm, void* parmhandle, int*
piLength, int* rgiInd);
```

Returns the length of a specific occurrence of a specific array parameter in a parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
piLength	Pointer to an int in which the length of the value is returned.
rgiInd	Three dimensional array of int values, indicating a specific array occurrence. The indices start with 0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	

nni_put_parm

Syntax

```
int nni_put_parm(void* pnni_func, int iParm, void* parmhandle, int iBufferLength,
const void* pBuffer);
```

Assigns a value to a specific parameter in a parameter set. The value is passed to the function in the buffer at the address specified in `pBuffer`, with the size specified in `iBufferLength`. See ***Natural Data Types*** on how to prepare the contents of the buffer.

If the length of the parameter according to the Natural data type is smaller than the given buffer length, the data will be truncated to the length of the parameter. The rest of the buffer will be ignored. If the length of the parameter according to the Natural data type is greater than the given buffer length, the data will copied only to the given buffer length, the rest of the parameter value stays unchanged. See ***Natural Data Types*** on the internal length of Natural data types.

If the parameter is a dynamic variable, it is automatically resized according to the given buffer length.

If the parameter is an array, the function expects the whole array in the buffer. This makes sense only for fixed size arrays of fixed size elements, because in other cases the caller cannot provide the correct contents of the buffer. In order to assign a value to an individual occurrence of an arbitrary array use the function nni_put_parm_array.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
iBufferLength	Length of the buffer specified in pBuffer.
pBuffer	Buffer in which the value is passed.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
= *n*, where $n > 0$	Successful operation, but only *n* bytes of the buffer were used.

nni_put_parm_array

Syntax

```
int nni_put_parm_array(void* pnni_func, int iParm, void* parmhandle, int
iBufferLength, const void* pBuffer, int* rgiInd);
```

Assigns a value to a specific occurrence of a specific array parameter in a parameter set. The only difference to nni_get_parm is that array indices can be specified. The indices for unused dimensions must be specified as 0.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
iBufferLength	Length of the buffer specified in pBuffer.
pBuffer	Buffer in which the value is passed.
rgiInd	Three dimensional array of int values, indicating a specific array occurrence. The indices start with 0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	
= *n*, where *n* > 0	Successful operation, but only *n* bytes of the buffer were used.

nni_resize_parm_array

Syntax

```
int nni_resize_parm_array(void* pnni_func, int iParm, void* parmhandle, int* rgiOcc);
```

Changes the occurrence count of a specific x-array parameter in a parameter set. For an *n*-dimensional array an occurrence count must be specified for all *n* dimensions. If the dimension of the array is less than 3, the value 0 must be specified for the not used dimensions.

The function tries to resize the occurrence count of each dimension either by changing the lower bound or the upper bound, whatever is appropriate for the given x-array.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
rgiOcc	Three dimensional array of int values, indicating the new occurrence count of the array.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
NNI_RC_NOT_ARRAY	
NNI_RC_NOT_RESIZABLE	
> 0	Natural error number.

nni_delete_parm

Syntax

```
int nni_delete_parm(void* pnni_func, void* parmhandle);
```

Deletes the specified parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
parmhandle	Pointer to a parameter set.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	

nni_from_string

Syntax

```
int nni_from_string(void* pnni_func, const char* szString, char chFormat, int
iLength, int iPrecision, int iBufferLength, void* pBuffer);
```

Converts the string representation of a Natural P, N, D or T value into the internal representation of the value, as it is used in the functions `nni_get_parm`, `nni_get_parm_array`, `nni_put_parm` and `nni_put_parm_array`.

The string representations of these Natural data types look like this:

Format	String representation
P, N	E. g. "-3.141592", where the decimal character defined in the `DC` parameter is used.
D	Date format as defined in the `DTFORM` parameter, (e. g. „2004-07-06“, if `DTFORM=I`).
T	Date format as defined in the `DTFORM` parameter, combined with a Time value in the form hh:ii:ss:t (e. g. "2004-07-06 11:30:42:7", if `DTFORM=I`) or Time value in the form hh:ii:ss:t (e. g. "11:30:42:7").

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szString	String representation of the value.
chFormat	Natural data type of the value.
iLength	Natural length of the value. The total number of significant digits in the case of NNI_TYPE_NUM and NNI_TYPE_PACK, 0 otherwise.
iPrecision	Number of decimal places in the case of NNI_TYPE_NUM and NNI_TYPE_PACK, 0 otherwise.
iBufferLength	Length of the buffer provided in pBuffer.
pBuffer	Buffer that contains the internal representation of the value on return. The buffer must be large enough to hold the internal Natural representation of the value. The required sizes are documented in ***Format and Length of User-Defined Variables***.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number

nni_to_string

Syntax

```
int nni_to_string(void* pnni_func, int iBufferLength, const void* pBuffer, char
chFormat, int iLength, int iPrecision, int iStringLength, char* szString);
```

Converts the internal representation of a Natural P, N, D or T value, as it is used in the functions nni_get_parm, nni_get_parm_array, nni_put_parm and nni_put_parm_array, into a the string representation.

The string representations of these Natural data types look as described with the function nni_from_string.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iBufferLength	Length of the buffer provided in pBuffer.
pBuffer	Buffer that contains the internal representation of the value. The required sizes are documented in ***Format and Length of User-Defined Variables***.
chFormat	Natural data type of the value.
iLength	Natural length of the value. The total number of significant digits in the case of NNI_TYPE_NUM and NNI_TYPE_PACK, 0 otherwise.
iPrecision	Number of decimal places in the case of NNI_TYPE_NUM and NNI_TYPE_PACK, 0 otherwise.
iStringLength	Length of the string buffer provided in szString including the terminating zero.
szString	String buffer that contains the string representation of the value on return. The string buffer must be large enough to hold the external representation including the terminating zero.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number

127 Parameter Description Structure

The interface provides information about the parameters of a Natural subprogram or method in a structure named `parameter_description`. The structure is defined in the header file *natuser.h*. This file is contained in the directory *%NATDIR%\%NATVERS%\samples\sysexnni*.

An array of `parameter_description` structures is passed to the interface with each call to `nni_callnat` and similar functions. A `parameter_description` structure is created from a parameter in a parameter set using the function `nni_get_parm_info`.

The relevant elements of the structure contain the following information. All elements not listed in this table are for internal use only.

Format	Element Name	Content
void*	address	Address of the parameter value. Must not be reallocated or freed. The address element is a null pointer for arrays of dynamic variables and for x-arrays. In these cases, the array data cannot be accessed as a whole, but can only be accessed elementwise through the parameter access function `nni_get_parm`.
int	format	Natural data type of the parameter. Refer to ***Natural Data Types*** for further information.
int	length	Natural length of the parameter value. In the case of the data types `NNI_TYPE_ALPHA` and `NNI_TYPE_UNICODE`, the number of characters. In the case of the data types `NNI_TYPE_PACK` and `NNI_TYPE_NUM`, the number of digits before the decimal character. In the case of an array, the length of a single occurrence. In the case of an array of dynamic variables, the length is indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.
int	precision	In the case of the data types `NNI_TYPE_PACK` and `NNI_TYPE_NUM` the number of digits after the decimal character, 0 otherwise.
int	byte_length	Length of the parameter value in bytes. In the case of an array the byte length of a single occurrence. In the case of an array of dynamic variables the byte length is

Format	Element Name	Content
		indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.
int	dimensions	Number of dimensions. 0 in the case of a scalar. The maximum number of dimensions is 3.
int	length_all	Total length of the parameter value in bytes. In the case of an array the byte length of the whole array. In the case of an array of dynamic variables the total length is indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.
int	flags	Parameter flags, see ***Flags***.
int	occurrences[10]	Number of occurrences in each dimension. Only the first three occurrences are used.
int	indexfactors[10]	Array indexfactors for each dimension. Only the first three occurrences are used.

In the case of arrays with fixed bounds of variables with fixed length, the array contents can be accessed directly using the structure element `address`. In these cases the following applies:

- The address of the element (i,j,k) of a three dimensional array is computed as follows:

  elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * indexfactors[2]

- The address of the element (i,j) of a two dimensional array is computed as follows:

  elementaddress = address + i * indexfactors[0] + j * indexfactors[1]

- The address of the element (i) of a one dimensional array is computed as follows:

  elementaddress = address + i * indexfactors[0]

128 Natural Data Types

Some of the parameter access functions (like `nni_get_parm`, `nni_put_parm`) use a buffer that contains a parameter value in the correct representation. The length of the buffer depends on the Natural data type. The data format of the buffer is defined according to the following table:

Natural Data Type	Buffer Format
A	char[ ]
B	byte[ ]
C	short
F4	float
F8	double
I1	signed char
I2	short
I4	int
L	`NNI_L_TRUE` or `NNI_L_FALSE`, see *natni.h*
HANDLE OF OBJECT	byte[8]
P, N, D, T	The buffer content should be created from a string representation with the function `nni_from_string`. It can be transformed to a string representation with the function `nni_to_string`.
U	An array of UTF-16 characters. On Windows and on those UNIX platforms where a wchar corresponds to a UTF-16 character, this is a wchar[].

Some of the parameter access functions (like `nni_get_parm`, and `nni_put_parm`) require a Natural data type to be specified. In these cases the following constants should be used. The constants are defined in the header file *natni.h*. This file is contained in the directory *%NATDIR%\%NATVERS%\samples\sysexnni*.

Natural Data Type	Constant
A	NNI_TYPE_ALPHA
B	NNI_TYPE_BIN
C	NNI_TYPE_CV
D	NNI_TYPE_DATE
F	NNI_TYPE_FLOAT
I	NNI_TYPE_INT
L	NNI_TYPE_LOG
N	NNI_TYPE_NUM
HANDLE OF OBJECT	NNI_TYPE_OBJECT
P	NNI_TYPE_PACK
T	NNI_TYPE_TIME
U	NNI_TYPE_UNICODE

129 Flags

The structure parameter_description has an element flags that contains information about the status of the parameter. Also the functions nni_init_parm* allow specifying some of these flags whe initializing a parameter. The individual flags can be combined with a logical OR in the element flags. The following flags are defined in the header file *natni.h*. This file is contained in the directory *%NATDIR%\%NATVERS%\samples\sysexnni*.

Return Code	Meaning
NNI_FLG_PROTECTED	Parameter is write protected.
NNI_FLG_DYNAMIC (*)	Parameter is dynamic (variable length or x-array).
NNI_FLG_NOT_CONTIG (*)	Array is not contiguous.
NNI_FLG_AIV (*)	Parameter is an AIV or INDEPENDENT variable.
NNI_FLG_DYNVAR (*)	Parameter has variable length.
NNI_FLG_XARRAY (*)	Parameter is an x-array.
NNI_FLG_LBVAR_0	Lower bound of dimension 0 is variable.
NNI_FLG_UBVAR_0	Upper bound of dimension 0 is variable.
NNI_FLG_LBVAR_1	Lower bound of dimension 1 is variable.
NNI_FLG_UBVAR_1	Upper bound of dimension 1 is variable.
NNI_FLG_LBVAR_2	Lower bound of dimension 2 is variable.
NNI_FLG_UBVAR_2	Upper bound of dimension 2 is variable.

Only the flags marked with "(*)" can be explicitly set in the functions nni_init_parm*. The other flags are automatically set by the interface according to the type of the parameter.

If one of the NNI_FLG_*VAR* flags is set, the array is an x-array. In each dimension of an x-array only the lower bound or the upper bound, not both, can be variable. Therefore for instance the flag NNI_FLG_LBVAR_0 may not be combined with NNI_FLG_UBVAR_0.

If NNI_FLG_DYNAMIC is on, also NNI_FLG_DYNVAR, NNI_FLG_XARRAY or both are on. If both are on, the parameter is an x-array with elements of variable length.

130 Return Codes

The interface functions return the following return codes. The constants are defined in the header file *natni.h*. This file is contained in the directory *%NATDIR%\%NATVERS%\samples\sysexnni*.

Return Code	Meaning
NNI_RC_OK	Successful execution.
NNI_RC_ILL_PNUM	Invalid parameter number.
NNI_RC_INT_ERROR	Internal error.
NNI_RC_DATA_TRUNC	Data has been truncated during parameter value access.
NNI_RC_NOT_ARRAY	Parameter is not an array.
NNI_RC_WRT_PROT	Parameter is write protected.
NNI_RC_NO_MEMORY	Memory allocation failed.
NNI_RC_BAD_FORMAT	Invalid Natural data type.
NNI_RC_BAD_LENGTH	Invalid length or precision.
NNI_RC_BAD_DIM	Invalid dimension count.
NNI_RC_BAD_BOUNDS	Invalid x-array bound definition.
NNI_RC_NOT_RESIZABLE	Array cannot be resized in the requested way.
NNI_RC_BAD_INDEX_0	Index for array dimension 0 out of range.
NNI_RC_BAD_INDEX_1	Index for array dimension 1 out of range.
NNI_RC_BAD_INDEX_2	Index for array dimension 2 out of range.
NNI_RC_VERSION_ERROR	Requested interface version not supported.
NNI_RC_NOT_INIT	No Natural session initialized in this interface instance.
NNI_RC_NOT_IMPL	Function not implemented in this interface version.
NNI_RC_PARM_ERROR	Mandatory parameter not specified.
NNI_RC_LOCKED	Interface instance is locked by another thread.
rc, where *rc* < NNI_RC_SERR_OFFSET	Natural startup error occurred. The Natural startup error number as documented in *Natural Startup Errors* (which is part of the

Return Code	Meaning
	Operations documentation) can be determined from the return code by the following calculation: `startup-error-nr = -(rc - NNI_RC_SERR_OFFSET)`
> 0	Natural error number.

131 Natural Exception Structure

The interface functions that execute Natural code (such as `nni_callnat`) return a structure named `natural_exception` that contains further information about a Natural error that might have occurred. The structure is defined in the header file *natni.h*. This file is contained in the directory *%NATDIR%\%NATVERS%\samples\sysexnni*.

The elements of the structure contain the following information.

Format	Element Name	Content
int	natMessageNumber	Natural message number.
char	natMessageText[NNI_LEN_TEXT+1]	Natural message text with all replacements.
char	natLibrary[NNI_LEN_LIBRARY+1]	Natural library name.
char	natMember[NNI_LEN_MEMBER+1]	Natural member name.
char	natName[NNI_LEN_NAME+1];	Natural function, subroutine or class name.
char	natMethod[NNI_LEN_NAME+1];	Natural method or property name.
int	int natLine;	Natural code line where the error occurred.

132 Interface Usage

The interface is typically used in the following way (example: Call a Natural subprogram):

1. Determine the location of the Natural binaries.
2. Load the Natural Native Interface library.
3. Call `nni_get_interface` to retrieve an interface instance.
4. Call `nni_initialize` to initialize a Natural session.
5. Call `nni_logon` to logon to a specific Natural library.
6. Call `nni_create_parm` or a related function to create a set of parameters.
7. For each parameter
   - Call one of the `nni_init_parm` functions to initialize the parameter to the correct type.
   - Call one of the `nni_put_parm` functions to assign a value to the parameter.
   - Call `nni_get_parm_info` to create the `parameter_description` structure.
8. Call `nni_callnat` to call the subprogram.
9. For each modifiable parameter.
   - Call one of the `nni_get_parm` functions to retrieve the parameter value.
10. Call `nni_delete_parm` to to free the parameter structures.
11. Call `nni_uninitialize` to uninitialize the Natural session.
12. Call `nni_logoff` to return to the previous library.
13. Call `nni_free_interface` to free the interface instance.

An example C program *nnisample.c* that shows the usage of the interface is contained in *%NAT-DIR%\%NATVERS%\samples\sysexnni*.

133 Threading Issues

A Natural process on Windows and UNIX always contains only one thread that executes Natural code. Thus in an interactively started Natural session, it can never occur that several threads try to execute Natural code in parallel. The situation is different when a client program that runs several threads in parallel uses the Natural Native Interface.

The Natural Native Interface can be used by multithreaded applications. The interface functions are thread safe. As long as a given thread T is executing one of the interface functions, other threads of the same process that call one of the interface functions are blocked until T has left the interface function. Effectively the parallel executing threads of the process are serialized as far as the usage of the interface functions is concerned. It is not necessary to serialize interface access among the threads of different processes, because each different process that uses the NNI runs its own Natural session.

The calling application can also control the multithreaded access to the NNI explicitly. This can make sense if a thread wants to execute a series of NNI calls without being interrupted by another thread. To achieve this, the thread calls `nni_enter`, which lets the thread wait until all other threads have left the NNI. Then the thread does its work and calls NNI functions at will. After having finished its work, the thread calls `nni_leave` to allow other threads to access the NNI.

A multithreaded application that uses the NNI must follow these rules:

- The functions `nni_initialize` and `nni_uninitialize` must be called at least once per process.
- The function `nni_uninitialize` must be called on the same thread as the corresponding call to `nni_initialize`.
- The function `nni_uninitialize` must not be called before the last thread that uses the NNI has terminated.

134 NaturalX

Anmerkung: Dieser Teil liegt nur teilweise in deutscher Sprache vor.

Dieser Teil beschreibt, wie objektbasierte Anwendungen entwickelt und verteilt werden.

- Einführung in NaturalX
- NaturalX-Anwendungen entwickeln
- Distributing NaturalX Applications
- ActiveX Component SoftwareAG.NaturalX.Utilities
- Interface INaturalXUtilities
- Interface IRunningObjects
- ActiveX Component SoftwareAG.NaturalX.Enumerator
- Interface IEnumerator

135 Einführung in NaturalX

- Warum NaturalX? 948
- Programming Techniques 949

Dieses Kapitel enthält eine kurze Einführung in die komponentenbasierte Programmierung, und damit einhergehend die Benutzung der NaturalX-Schnittstelle und einem speziell für diesen Zweck vorgesehenen Satz Natural-Statements.

Warum NaturalX?

Auf Komponenten-Architektur basierende Software-Anwendungen bieten viele Vorteile gegenüber traditionellen Designs. Diese sind u.a. die Folgenden:

- Schnellere Entwicklung. Die Programmierer können Anwendungen schneller erstellen, indem sie die Software aus vorerstellten Komponenten zusammensetzen.
- Geringere Entwicklungskosten. Eine allgemeine Menge von Schnittstellen für Programme zur Verfügung zu haben, bedeutet weniger Arbeit bei der Integration der Komponenten in vollständige Lösungen.
- Höhere Flexibilität. Es ist leichter, Software für unterschiedliche Abteilungen innerhalb eines Unternehmens zu standardisieren, indem Sie einfach einige der Komponenten verändern, aus denen die Anwendung besteht.
- Reduzierte Wartungskosten. Im Falle eines Umstiegs auf eine neue Version ist es häufig ausreichend, einige der Komponenten zu ändern, anstatt die gesamte Anwendung ändern zu müssen.
- Einfachere Verteilung. Komponenten enthalten gekapselte Datenstrukturen und Funktionalität in verteilt installierbaren Einheiten.

Mit NaturalX können Sie komponentenbasierte Anwendungen erstellen.

Sie können NaturalX in Verbindung mit DCOM einsetzen. Das bietet folgende Möglichkeiten:

- Andere Komponenten können auf die von Ihnen entwickelten Komponenten zugreifen.
- Sie können diese Komponenten auf lokalen und/oder Remote-Servern ausführen.
- Sie können aus Natural-Programmen über Prozess- und Maschinengrenzen hinweg auf Komponenten zugreifen, die in den verschiedensten Programmiersprachen geschrieben wurden.
- Sie können Ihre vorhandenen Natural-Anwendungen mit (Quasi-) Standard-Schnittstellen ausstatten.

Das nachfolgend beschriebene Szenario demonstriert, wie ein Unternehmen aus diesen Vorteilen Nutzen ziehen kann. Ein Unternehmen führt ein Vertriebsverwaltungssystem ein, das auf einem Anwendungskonzept auf der Basis von Komponenten beruht. In dieser Anwendung sind für jede Vertriebsstelle zahlreiche Datenerfassungskomponenten vorhanden. Alle diese Vertriebsstellen benutzen jedoch eine gemeinsame Steuerberechnungskomponente, die auf einem Server läuft. Bei Änderungen in der Steuergesetzgebung muss man nicht die Datenerfassungskomponenten an jedem einzelnen Standort ändern, sondern braucht lediglich die Steuer-Komponente zu aktualisieren. Ausserdem haben es die Programmierer leichter, weil sie sich nicht mehr den Kopf

über das Programmieren von Netzwerken und die Integration von in unterschiedlichen Sprachen geschriebenen Komponenten zerbrechen müssen.

Programming Techniques

This section covers the following topics:

- Object-Based Programming
- Defining Classes
- Defining Interfaces
- Interface Inheritance

Object-Based Programming

NaturalX follows an object-based programming approach. Characteristic for this approach is the encapsulation of data structures with the corresponding functionality into classes. Encapsulation is a good basis for easy distribution. Because there are (quasi) standards for the interoperation of software components on the basis of object models, an object-based approach is also a good basis for making software components interoperable across program, machine and programming language boundaries.

Defining Classes

In an object-based application, each function is considered to be a service that is provided by an object. Each object belongs to a class. Clients use the services either to perform a business task or to build even more complex services and to provide these to other clients. Hence the basic step in creating an application with NaturalX is to define the classes that form the application. In many cases, the classes simply correspond to the real things that the application in question deals with, for example, bank accounts, aircraft, shipments etc. There is a wide range of good literature about object-oriented design, and a number of well-proven methods can be used to identify the classes in a given business.

The process of defining a class can be broadly broken down into the following steps:

- Create a Natural module of type class.
- Specify the name of the class using the `DEFINE CLASS` statement. This name will be used by the clients to create objects of that class.
- Use the `OBJECT` clause of the `DEFINE DATA` statement to define how an object of the class will look internally. Create a local data area that describes the layout of the object with the data area editor, and assign this data area in the `OBJECT` clause.

These steps are described in more detail in the section ***Developing Object-Based Natural Applications***.

Defining Interfaces

In order to be useful to clients, a class must provide services, which it does through interfaces. An interface is a collection of methods and properties. A method is a function that an object of the class can perform when requested by a client. A property is an attribute of an object that a client can retrieve or change. A client accesses the services by creating an object of the class and using the methods and properties of its interfaces.

The process of defining an interface can be broadly broken down into the following steps:

- Use the `INTERFACE` clause to specify an interface name.
- Define the properties of the interface with `PROPERTY` definitions.
- Define the methods of the interface with `METHOD` definitions.

These steps are described in more detail in the section ***Developing Object-Based Natural Applications***.

Simple classes only have one interface, but a class may have more than one interface. This possibility can be used to group methods and properties into one interface that belong to the same functional aspect of the class and to define different interfaces to handle other functional aspects. For example, an `Employee` class could have an interface `Administration` that contains all of the methods and properties of the administrative aspects of an employee. This interface could contain the properties `Salary` and `Department` and the method `TransferToDepartment`. Another interface `Qualifications` could contain the qualification aspects of an employee.

Interface Inheritance

Defining several interfaces for a class is the first step towards using interface inheritance, which is a more advanced method of designing classes and interfaces. This makes it possible to reuse the same interface definition in different classes. Assume that there is a class `Manager`, which is to be treated in the same way as the class `Employee` with respect to qualification, but which is to be handled differently as far as administration is concerned. This can be achieved by having the `Qualification` interface in both classes. This has the advantage that a client that uses the `Qualification` interface on a given object does not have to check explicitly whether the object represents an `Employee` or a `Manager`. It can simply use the same methods and properties without having to know of what class the object is. The properties or methods can even be implemented in a different way in both classes provided they are presented through the same interface definition.

The process of using interface inheritance can be broadly broken down into the following steps:

- Use the `INTERFACE` statements to define one or more interfaces in a copycode instead of defining them directly in the class.
- The `METHOD` and `PROPERTY` definitions in the `INTERFACE` statement do not need to contain the `IS` clause. At this point, you just define the external appearance of the interface without assigning implementations to the methods and properties.

- Use the `INTERFACE` clause to include the copycode with its interface definition in each class that will implement the interface.
- Use the `METHOD` and `PROPERTY` statements to assign implementations to the methods and properties of the interface in each class that will implement the interface.

136 NaturalX-Anwendungen entwickeln

- Entwickungsumgebungen 954
- Klassen definieren 954
- Klassen und Objekte benutzen 959

Dieses Dokument beschreibt, wie eine Anwendung durch Definition und Benutzung von Klassen entwickelt wird.

Entwickungsumgebungen

- **Klassen auf Windows-Plattformen entwickeln**
  Auf Windows-Plattformen stellt Natural den Class Builder als Werkzeug zur Entwicklung von Natural-Klassen zur Verfügung. Der Class Builder präsentiert eine Natural-Klasse in einer strukturierten hierarchischen Reihenfolge und ermöglicht es dem Benutzer, die Klassen und ihre Komponenten effizient zu verwalten. Wenn Sie den Class Builder benutzen, sind überhaupt keine Vorkenntnisse oder auch nur Grundkenntnisse der unten beschriebenen Syntax-Elemente erforderlich.
- **Klassen mit SPoD entwickeln**
  In einer Natural Single Point of Development-Umgebung (SPoD), der zentralen Umgebung für Entwicklungen einschließlich eines Großrechners und/oder UNIX Remote Development Servers (DFÜ-Entwicklungsserver) können Sie den Class Builder benutzen, der zusammen mit der Natural Studio-Benutzeroberfläche zur Entwicklung von Klassen auf Großrechnern und/oder UNIX-Plattformen zur Verfügung steht. In diesem Fall sind keine Vorkenntnisse oder auch nur Grundkenntnisse der im Folgenden beschriebenen Syntax-Elemente erforderlich.
- **Klassen auf Großrechner- oder UNIX-Plattformen entwickeln**
  Wenn Sie SpoD nicht benutzen, entwickeln Sie mittels des Natural-Programm-Editors Klassen auf diesen Plattformen. In diesem Fall sollten Sie die Syntax der unten beschriebenen Klassen-Definition kennen.

Klassen definieren

Wenn Sie eine Klasse definieren, müssen Sie ein Natural-Klassenmodul definieren, innerhalb dessen Sie ein `DEFINE CLASS`-Statement erstellen. Mittels des `DEFINE CLASS`-Statements können Sie der Klasse einen extern verwendbaren Namen zuweisen und ihre Schnittstellen, Methoden und Eigenschaften definieren. Der Klasse kann auch ein Objekt-Datenbereich zugewiesen werden, der das Layout einer Instanz der Klasse beschreibt. Das `DEFINE CLASS`-Statement dient außerdem dazu, einen Global Unique Identifier für diejenigen Klassen zur Verfügung zu stellen, die als COM-Klassen registriert werden sollen.

Dieser Abschnitt umfasst die folgenden Themen:

- Natural-Klassenmodul erstellen
- Klasse spezifizieren
- Schnittstelle definieren
- Objekt-Datenvariable einer Property zuweisen
- Subprogramm einer Methode zuweisen

- Methoden implementieren

Natural-Klassenmodul erstellen

Um ein Natural-Klassenmodul zu erstellen

- Benutzen Sie das `CREATE OBJECT`-Statement zur Erstellung eines Natural-Objekts des Typs Klasse.

Klasse spezifizieren

Das `DEFINE CLASS`-Statement definiert den Namen der Klasse, die Schnittstellen, die die Klasse unterstützt, und die Struktur ihrer Objekte. Das `DEFINE CLASS`-Statement dient außerdem dazu, einen Global Unique Identifier und die Activation Policy für diejenigen Klassen zur Verfügung zu stellen, die als COM-Klassen registriert werden sollen.

Um eine Klasse zu spezifizieren

- Benutzen Sie das `DEFINE CLASS`-Statement, das in der *Statements*-Dokumentation beschrieben ist.

Schnittstelle definieren

Jede Schnittstelle einer Klasse wird mit einem `INTERFACE`-Statement innerhalb der Klassen-Definition angegeben. Ein `INTERFACE`-Statement gibt den Namen der Schnittstelle und eine Anzahl von Eigenschaften und Methoden an. Für Klassen, die als COM–Klassen registriert werden sollen, gibt es auch die Globally Unique ID der Schnittstelle an.

Eine Klasse kann eine oder mehrere Schnittstellen haben. Für jede Schnittstelle wird ein `INTERFACE`-Statement in der Klassen-Definition kodiert. Jedes `INTERFACE`-Statement enthält eine oder mehrere `PROPERTY`- und `METHOD`-Klauseln. Gewöhnlich stehen die in einer Schnittstelle enthaltenen Eigenschaften und Methoden miteinander in einem technischen oder betriebswirtschaftlichen Zusammenhang.

Die `PROPERTY`-Klausel definiert den Namen einer Eigenschaft und weist der Eigenschaft eine Variable vom Objekt-Datenbereich zu. Diese Variable wird zum Speichern des Wertes der Eigenschaft benutzt.

Die `METHOD`-Klausel definiert den Namen einer Methode und weist der Methode ein Subprogramm zu. Dieses Subprogramm wird zum Implementieren der Methode benutzt.

Um eine Schnittstelle zu definieren

- Benutzen Sie das `INTERFACE`-Statement (siehe *Statements*-Dokumentation).

Objekt-Datenvariable einer Property zuweisen

Das PROPERTY-Statement wird nur benutzt, wenn mehrere Klassen dieselbe Schnittstelle auf verschiedene Art und Weise implementieren sollen. In diesem Fall benutzen die Klassen dieselbe Schnittstellen-Definition gemeinsam und beziehen sie von einem Natural-**Copycode** ein. Das PROPERTY-Statement wird dann benutzt, um *außerhalb* der Schnittstellen-Definition eine Variable vom Objekt-Datenbereich einer Eigenschaft zuzuweisen. Wie die PROPERTY-Klausel des INTERFACE-Statements definiert das PROPERTY-Statement den Namen einer Eigenschaft und weist eine Variable vom Objekt-Datenbereich der Eigenschaft zu. Diese Variable wird zum Speichern des Wertes der Eigenschaft benutzt.

Um eine Objekt-Datenvariable einer Eigenschaft zuzuweisen

- Benutzen Sie das PROPERTY-Statement (siehe *Statements*-Dokumentation).

Subprogramm einer Methode zuweisen

Das METHOD-Statement wird nur benutzt, wenn mehrere Klassen dieselbe Schnittstelle auf verschiedene Arten implementieren sollen. In diesem Fall benutzen die Klassen dieselbe Schnittstellen-Definition gemeinsam und beziehen sie von einem Natural-Copycode ein. Das METHOD-Statement wird dann benutzt, um außerhalb der Schnittstellen-Definition der Methode ein Subprogramm zuzuweisen. Wie die METHOD-Klausel des INTERFACE-Statements definiert das METHOD-Statement den Namen einer Methode und weist der Methode ein Subprogramm zu. Dieses Subprogramm wird zum Implementieren der Methode benutzt.

Um einer Methode ein Subprogramm zuzuweisen

- Benutzen Sie das METHOD-Statement (siehe *Statements*-Dokumentation).

Methoden implementieren

Eine Methode wird in der folgenden allgemeinen Form als ein Natural-Subprogramm implementiert:

```
DEFINE DATA
*
* Implementation code of the method
*
END
```

Informationen zum DEFINE DATA-Statement siehe *Statements*-Dokumentation.

Alle Klauseln des DEFINE DATA-Statements sind optional.

Um die Daten-Konsistenz sicherzustellen, empfiehlt es sich, dass Sie keine Inline-Datendefinitionen, sondern Data Areas benutzen.

Wenn eine `PARAMETER` clause-Klausel angegeben wird, kann die Methode Parameter und/oder einen Rückgabewert haben.

Mit `BY VALUE` in der Parameter Data Area markierte Parameter sind Eingabe-Parameter der Methode.

Nicht mit `BY VALUE` markierte Parameter werden „By Reference“ übergeben und sind Eingabe/Ausgabe-Parameter. Dies ist die Voreinstellung.

Der erste mit `BY VALUE RESULT` markierte Parameter wird als Rückgabewert für die Methode zurückgegeben. Wenn mehr als ein Parameter auf diese Art markiert wird, werden die anderen als Eingabe/Ausgabe-Parameter behandelt.

Als `OPTIONAL` markierte Parameter brauchen nicht angegeben zu werden, wenn die Methode aufgerufen wird und Sie die *n*`X`-Notation im `SEND METHOD`-Statement benutzen.

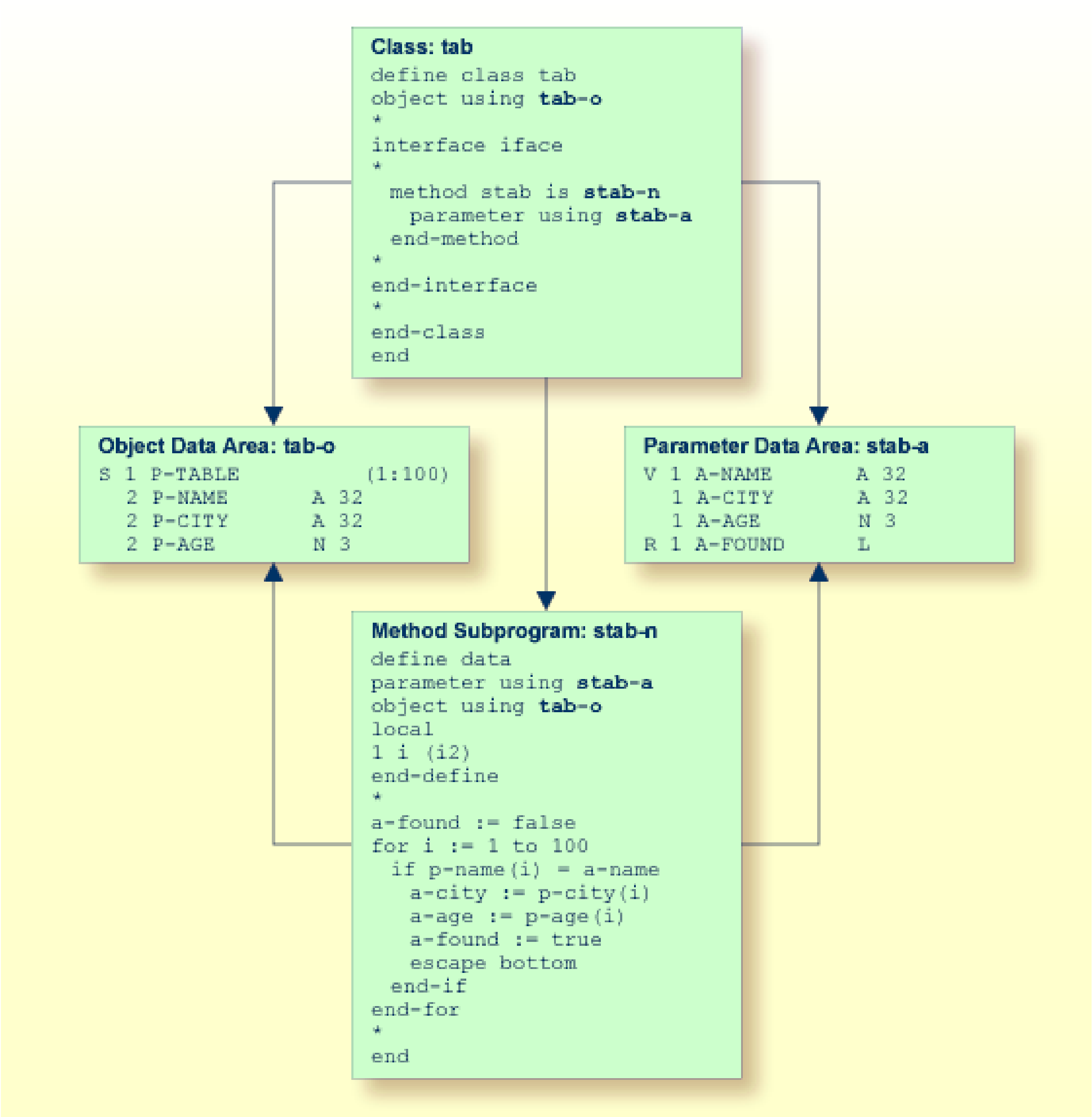
Um sicherzustellen, dass das Methoden-Subprogramm genau dieselben Parameter akzeptiert, wie in dem entsprechenden `METHOD`-Statement in der Klassen-Definition angegeben, benutzen Sie eine Parameter Data Area anstatt von Inline-Datendefinitionen. Benutzen Sie dieselbe Parameter Data Area wie in dem betreffenden `METHOD`-Statement.

Um dem Methoden-Subprogramm Zugriff auf die Objektdaten-Struktur zu geben, kann die `OBJECT`-Klausel angegeben werden. Um sicherzustellen, dass das Methoden-Subprogramm korrekt auf die Objektdaten zugreifen kann, benutzen Sie eine Local Data Area anstatt von Inline-Datendefinitionen. Verwenden Sie dieselbe Local Data Area, wie in der `OBJECT`-Klausel des `DEFINE CLASS`-Statements angegeben.

Die Klauseln `GLOBAL`, `LOCAL` und `INDEPENDENT` können wie in jedem anderen Natural-Programm benutzt werden.

Obwohl es technisch möglich ist, macht es gewöhnlich keinen Sinn, eine `CONTEXT`-Klausel in einem Methoden-Subprogramm zu benutzen.

In dem folgenden Beispiel werden Daten über eine vorgegebene Person von einer Tabelle eingelesen. Der Suchschlüssel wird als ein `BY VALUE`-Parameter übergeben. Die sich daraus ergebenden Daten werden über “by reference”-Parameter zurückgegeben („By Reference“ ist die Standard-Definition). Der Rückgabewert der Methode wird durch die Spezifikation `BY VALUE RESULT` definiert.


Class: tab
define class tab
object using tab-o
*
interface iface
*
method stab is stab-n
parameter using stab-a
end-method
*
end-interface
*
end-class
end
Object Data Area: tab-o
S 1 P-TABLE (1:100)
2 P-NAME A 32
2 P-CITY A 32
2 P-AGE N 3
Parameter Data Area: stab-a
V 1 A-NAME A 32
1 A-CITY A 32
1 A-AGE N 3
R 1 A-FOUND L
Method Subprogram: stab-n
define data
parameter using stab-a
object using tab-o
local
1 i (i2)
end-define
*
a-found := false
for i := 1 to 100
if p-name(i) = a-name
a-city := p-city(i)
a-age := p-age(i)
a-found := true
escape bottom
end-if
end-for
*
end

Klassen und Objekte benutzen

Auf in einer lokalen Natural-Session erstellte Objekte kann von anderen Modulen aus in derselben Natural-Session zugegriffen werden.

Auf Objekte, die in anderen Prozessen oder auf Remote-Maschinen erstellt wurden, kann über DCOM zugegriffen werden.

In beiden Fällen gelten dieselben Regeln für den Zugriff auf und die Verwendung von Klassen und ihren Objekten.

Das Statement `CREATE OBJECT` wird benutzt, um ein Objekt (auch als *Instance* bezeichnet) einer gegebenen Klasse zu erstellen.

Um Objekte in Natural-Programmen zu referenzieren, müssen Object-Handles im `DEFINE DATA`-Statement definiert werden. Methoden eines Objekts werden mit dem Statement `SEND METHOD` aufgerufen. Objekte können Eigenschaften haben, auf die mit der normalen Zuweisungssyntax zugegriffen wird.

Anmerkung: Damit Sie eine NaturalX-Klasse über DECOM benutzen können, müssen Sie sie zuerst registrieren.

Diese Schritte sind im Folgenden beschrieben:

- Objekt-Handles definieren
- Instanz einer Klasse erstellen
- Bestimmte Methode eines Objekts aufrufen
- Properties aufrufen
- Beispielanwendung

Objekt-Handles definieren

Um Objekte in Natural-Programmen zu referenzieren, müssen Object-Handles im `DEFINE DATA`-Statement wie folgt definiert werden:

```
DEFINE DATA
  level-handle-name [(array-definition)] HANDLE OF OBJECT
  ...
END-DEFINE
```

Beispiel:

```
DEFINE DATA LOCAL
1 #MYOBJ1 HANDLE OF OBJECT
1 #MYOBJ2 (1:5) HANDLE OF OBJECT
END-DEFINE
```

Instanz einer Klasse erstellen

Um eine Instanz einer Klasse zu erstellen

- Benutzen Sie das `CREATE OBJECT`-Statement (siehe Natural *Statements*-Dokumentation).

Bestimmte Methode eines Objekts aufrufen

Um eine bestimmte Methode eines Objekts aufzurufen

- Benutzen Sie das `SEND METHOD`-Statement (siehe Natural *Statements*-Dokumentation).

Properties aufrufen

Auf Properties kann mit dem Statement `ASSIGN` (oder `COMPUTE`) wie folgt zugegriffen werden:

```
ASSIGN operand1.property-name = operand2
ASSIGN operand2 = operand1.property-name
```

Object Handle — *`operand1`*

`Operand1` muss als eine Object-Handle definiert werden und identifiziert das Objekt, dessen Eigenschaft aufgerufen werden soll. Das Objekt muss bereits vorhanden sein.

`operand2`

Als *`operand2`* geben Sie einen Operanden an, dessen Format datenübertragungskompatibel zu dem Format der Eigenschaft sein muss. Weitere Informationen siehe ***Kompatibilitätsregeln zur Datenübertragung***.

Wenn auf das Objekt über DCOM zugegriffen werden soll, sind die Regeln zur Konvertierung von Datentypen im Abschnitt *Using Type Information* in der *Operations*-Dokumentation zu berücksichtigen.

`property-name`

Der Name einer Property des Objekts.

Stimmt der Name der Property mit der Natural Identifier Syntax überein, kann er wie folgt angegeben werden:

```
create object #o1 of class "Employee"
  #age := #o1.Age
```

Wenn der Name der Property nicht mit der Natural Identifier Syntax übereinstimmt, muss er in spitze Klammern gesetzt werden:

```
create object #o1 of class "Employee"
  #salary := #o1.<<%Salary>>
```

Der Name der Property kann auch mit einem Schnittstellen-Namen qualifiziert werden. Dies ist erforderlich, wenn das Objekt mehr als eine Schnittstelle hat, die eine Property mit demselben Namen enthält. In diesem Fall muss der qualifizierte Name der Property in spitzen Klammern stehen:

```
create object #o1 of class "Employee"
  #age := #o1.<<PersonalData.Age>>
```

Beispiel:

```
define data
  local
  1 #i          (i2)
  1 #o          handle of object
  1 #p          (5) handle of object
  1 #q          (5) handle of object
  1 #salary     (p7.2)
  1 #history    (p7.2/1:10)
  end-define
  * ...
  * Code omitted for brevity.
  * ...
  * Set/Read the Salary property of the object #o.
  #o.Salary := #salary
  #salary := #o.Salary
  * Set/Read the Salary property of
  * the second object of the array #p.
  #p.Salary(2) := #salary
  #salary := #p.Salary(2)
  *
  * Set/Read the SalaryHistory property of the object #o.
  #o.SalaryHistory := #history(1:10)
```

```
#history(1:10) := #o.SalaryHistory
* Set/Read the SalaryHistory property of
* the second object of the array #p.
#p.SalaryHistory(2) := #history(1:10)
#history(1:10) := #p.SalaryHistory(2)
*
* Set the Salary property of each object in #p to the same value.
#p.Salary(*) := #salary
* Set the SalaryHistory property of each object in #p
* to the same value.
#p.SalaryHistory(*) := #history(1:10)
*
* Set the Salary property of each object in #p to the value
* of the Salary property of the corresponding object in #q.
#p.Salary(*) := #q.Salary(*)
* Set the SalaryHistory property of each object in #p to the value
* of the SalaryHistory property of the corresponding object in #q.
#p.SalaryHistory(*) := #q.SalaryHistory(*)
*
end
```

Um Arrays von Objekt-Handles und Eigenschaften mit Arrays als Werte korrekt benutzen zu können, ist es wichtig, Folgendes zu wissen:

Eine Eigenschaft einer Array-Ausprägung von Objekt-Handles wird mit der folgenden Index-Notation adressiert:

```
#p.Salary(2) := #salary
```

Auf eine Eigenschaft, die ein Array als Wert hat, wird stets als Ganzes zugegriffen. Deshalb ist keine Index-Notation mit dem Namen der Eigenschaft erforderlich:

```
#o.SalaryHistory := #history(1:10)
```

Eine Eigenschaft einer Array-Ausprägung von Objekt-Handles, die ein Array als Wert hat, wird deswegen wie folgt adressiert:

```
#p.SalaryHistory(2) := #history(1:10)
```

Beispielanwendung

Eine Beispielanwendung finden Sie in den Libraries `SYSEXCOM` und `SYSEXCOC`. Die `A-README`-Objekte in diesen Libraries enthalten Information darüber, wie das Beispiel ausgeführt wird.

137 Distributing NaturalX Applications

- General 966
- Globally Unique Identifiers - GUIDs 968

An application consisting of NaturalX classes can be distributed across several processes and machines using DCOM.

See also *Using Statements and Commands in a NaturalX Server Environment* in the *Operations* documentation.

General

Using NaturalX, you can make Natural classes and their services available to local and remote clients, thus creating distributed applications. Local clients are processes that run on the same machine as a given NaturalX server, and remote clients are processes that run on a different machine.

In order to distribute applications, a widely used distributed object technology is used - the Microsoft Distributed Component Object Model (DCOM). When you register a Natural class to DCOM, its interfaces are presented to clients in a quasi-standardized fashion as dynamic COM interfaces, which are also known as dispatch interfaces. These interfaces can be easily addressed by many programming languages including Visual Basic, Java, C++ and, of course, Natural.

There are several points that must be taken into consideration when organizing the distribution of a NaturalX application. Each of these points is discussed in more detail in this section and in the *Operations* documentation.

- Determine whether each class should be internal, external or local (see the section ***Internal, External and Local Classes***).
- Globally unique IDs (GUIDs) must be assigned to the internal and external classes and their interfaces in order to be able to address them uniquely in the network (see the section ***Globally Unique Idenitfiers (GUIDs)***).
- You can define the activation policy for each class in order to control the conditions under which DCOM activates it (see section *Activation Policies* in the *Operations* documentation).
- In order to organize classes to applications, you can define NaturalX servers and assign the classes to them (see the section *NaturalX Servers* in the *Operations* documentation).
- Classes must be registered to make them known to DCOM (see section *Registration* in the *Operations* documentation).
- You can configure an application in order to further control its behavior (see the sections *Configuration Overview* and *DCOM Configuration on Windows* in the *Operations* documentation).

Internal, External and Local Classes

It is important to distinguish between classes for internal use, classes for external use and those for local use only.

Internal Classes

Objects (instances) of internal classes can only be created in the client process.

Internal classes have the following features:

- Access to client session-dependent resources such as files and system variables.
- Can run within the client transaction.
- Can be debugged using the Natural debugger (local debugging).

External Classes

Objects (instances) of external classes can be created in a different process or on a different machine. If the client process is simultaneously a server for the class, they can also be created in the client process.

External classes have the following features:

- No access to client session-dependent resources such as stacks, files and system variables.
- Do not run within the client transaction.
- Can be used by remote nodes.
- Can be used by various clients using a variety of languages such as Natural, Java, Visual Basic, C/C++, etc.
- Can be debugged with the Natural debugger (remote debugging).

Local Classes

Local classes are classes, which are executed in local execution mode. Natural executes a class locally (within the Natural session) if it is either not registered or if DCOM is not available.

Local classes have the following features:

- Can be used even if DCOM is not available.
- Need not be registered with DCOM.
- Cannot be used from outside the client process.

Globally Unique Identifiers - GUIDs

DCOM uses global unique identifiers (GUIDs) - 128-bit integers that are virtually guaranteed to be unique throughout the world - to identify every interface and every class. This helps to ensure that server components can be located and to prevent clients connecting to an object accidentally.

If a class is to be registered to DCOM, every interface defined in a Natural class and the class itself must be associated with such a globally unique ID.

Once a globally unique ID has been assigned to an interface or a class, the ID must never be changed.

Using the Class Builder

Natural provides the Class Builder as the tool to develop Natural classes. The Class Builder automatically assigns a GUID to every class and interface.

138 ActiveX Component SoftwareAG.NaturalX.Utilities

- Purpose 970
- Interfaces 970

Purpose

The ActiveX component `SoftwareAG.NaturalX.Utilities` provides a number of methods that are useful in the context of NaturalX and Natural Studio plug-ins.

As an example, the general usage of the component in a Natural application is in the following way.

```
define data
local
1 #util handle of object
1 #studio handle of object
end-define
*
* First create an instance of the class SoftwareAG.NaturalX.Utilities.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Now call the individual methods of the component, for instance
* to get access to the Natural Studio Automation Interface.
*
send 'GetThisNaturalStudio' to #util return #studio
if #studio eq null-handle
  escape routine
end-if
*
end
```

Interfaces

The individual interfaces, their methods and their usage are described in detail in separate documents.

The component provides the following interfaces:

- Interface INaturalXUtilities
- Interface IRunningObjects

139 Interface INaturalXUtilities

- Purpose 972
- Methods 972

Purpose

This is the main interface of the component `SoftwareAG.NaturalX.Utilities`. It is returned when a new instance of the component is created.

Example:

```
define data
local
1 #util handle of object
end-define
*
* Create an instance of the class SoftwareAG.NaturalX.Utilities.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
end
```

After successful execution of the `CREATE OBJECT` statement, the variable `#util` contains an interface of the type `INaturalXUtilities`.

Methods

The following methods are available:

- GetThisNaturalStudio
- GetRunningObjects
- BindToObject

GetThisNaturalStudio

Retrieves the root interface `INatAutoStudio` of the current Natural Studio session. After having retrieved this interface, the client has access to Natural Studio functionality as it is provided by the Natural Studio Automation interface.

Parameters

Name	Natural Type	Variant Type	Remark
Return value	HANDLE OF OBJECT	VT_DISPATCH (INatAutoStudio)	

Return Value
The root interface `INatAutoStudio` of the current Natural Studio session. `NULL-HANDLE`, if the method is called in a Natural session that is not a Natural Studio session.

GetRunningObjects

Returns an interface `IRunningObjects` that is used to iterate across the names of the objects contained in the running objects table (ROT).

Parameters

Name	Natural Type	Variant Type	Remark
Return value	HANDLE OF OBJECT	VT_DISPATCH (**IRunningObjects**)	

Return Value
An interface `IRunningObjects` that is used to iterate across the names of the objects contained in the running objects table (ROT).

BindToObject

Returns an interface to an object that is identified by a specific kind of name, a so-called „moniker“ (Windows terminology). See ***Interface IRunningObjects*** for the necessary information about monikers.

Parameters

Name	Natural Type	Variant Type	Remark
Return value	HANDLE OF OBJECT	VT_DISPATCH	
Name	A	VT_BSTR	By value

Return Value
An interface to the object identified by the name specified in `Name`.

Name
Used to identify a specific object by name. The name must be from one of the following categories:

- A file moniker, for instance *c:\MyDoc.doc*.
- An URL moniker, for instance *http://www.myorg.org/MyDoc.doc* or *ftp://ftp.myorg.org/MyDoc.doc*.

- A name of an object contained in the ROT. The names of the objects in the ROT can be retrieved using the interface `IRunningObjects`.

If a file or URL moniker is specified, the corresponding object is loaded into the application that is registered for the corresponding file extension and an interface pointer (object handle) to the object is returned. If the object is already loaded into the application, an interface pointer (object handle) to the already running instance is returned.

Example:

```
define data
local
1 #util handle of object
1 #obj handle of object
1 #content handle of object
1 #word handle of object
1 #doc (a) dynamic
1 #text (a) dynamic
end-define
*
* Create an instance of the utilities class.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Load a document into Microsoft Word.
* The option (ad=o) is essential, because the
* method expects a by value parameter.
#doc := 'c:\word.doc'
send 'BindToObject' to #util with #doc (ad=o) return #obj
if #obj eq null-handle
  escape routine
end-if
*
* Access the content of the document.
#content := #obj.Content
#text := #content.Text
write 'Content:' #text (al=60)
*
* Close Microsoft Word.
#word := #obj.Application
send 'Quit' to #word
*
end
```

140 Interface IRunningObjects

- Purpose 976
- Methods 978

Purpose

This interface is an iterator across the names of the objects that are contained in the running objects table (ROT). The ROT is a system table that is provided and maintained by Windows. It allows applications to make the objects or documents on which the user is currently working available to other applications.

Each object contained in this table is identified by a so-called „moniker". A moniker is a name that follows a specific syntax. For instance, there are file monikers that identify a file in the file system. A file moniker in its readable form is nothing else than a file name with full path name, like *c:\MyDoc.doc*. As another example, there are URL monikers that identify a resource in the internet and the protocol to be used to access it. A URL moniker in its readable form is just a common URL, like *http://www.myorg.org/MyDoc.doc*.

An application that wants to access an object in the ROT specifies the moniker that identifies the object and receives an interface pointer (an object handle) to the object.

Applications often enter the object or document on which the user is currently working in the ROT.

Example

If you open the document *c:\MyDoc.doc* in Microsoft Word, Microsoft Word will enter the name of this document (that is: *c:\MyDoc.doc*) and an interface pointer to this document into the ROT.

When a Natural Studio session is started, Natural Studio enters the Automation root interface `INatAutoStudio` of this session into the ROT. The interface is identified by a name built as follows:

```
NaturalStudio/<version>/<userid>/>processid>
```

Example

```
NaturalStudio/n.n/SCULLY/42
```

where `n.n` is the product version.

By specifying this name, other applications can retrieve the Automation root interface in the ROT and use it to access the Natural Studio session.

The interface `IRunningObjects` allows iterating across the names of all objects currently contained in the ROT. The found names can then be used in the method `INaturalXUtilities::BindToObject` to retrieve an interface pointer (object handle) to the corresponding object.

Example

In the following sample program, the characters *n.n* stand for the Natural product version. Please, replace these characters by the current Natural product version if you want to run the sample program.

```
define data
local
1 #util handle of object
1 #studio handle of object
1 #objects handle of object
1 #progs handle of object
1 #prog handle of object
1 #rot handle of object
1 #ro (a) dynamic
end-define
*
* Create an instance of the Utilities class.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Retrieve the running objects table.
send 'GetRunningObjects' to #util return #rot
if #rot eq null-handle
  escape routine
end-if
*
* Iterate across the running objects table.
repeat
  send 'Next' to #rot return #ro
  if #ro eq ' '
    escape bottom
  end-if
* If we hit a running Natural Studio session,...
  if substring(#ro,1,17) eq 'NaturalStudio/n.n'
*   ...we access it,...
    send 'BindToObject' to #util
    with #ro (ad=o) return #studio
*   ...open a Program Editor in that session...
    #objects := #studio.objects
    #progs := #objects.programs
    send 'Add' to #progs
    with 1009 return #prog
*   ...and display the identifier of this session in the editor.
    compress 'This is' #ro to #ro
    #prog.source := #ro
  end-if
end-repeat
```

```
*
end
```

Methods

The following methods are available:

- Next
- Reset

Next

Returns the name of the next object in the ROT. The name can then be used in the method `INaturalXUtilities::BindToObject` to retrieve an interface pointer (object handle) to the corresponding object.

Parameters

Name	Natural Type	Variant Type	Remark
Return value	A	VT_BSTR	

Return Value

The name of the next object in the ROT.

Reset

Resets the iterator to its initial state. After having called `Reset`, a subsequent call to `Next` returns the name of the first object in the ROT.

141 ActiveX Component SoftwareAG.NaturalX.Enumerator

- Purpose 980
- Interface 981

Purpose

The ActiveX component `SoftwareAG.NaturalX.Enumerator` provides a general enumerator class that can be used to iterate across collections of Automation objects.

As an example, the general usage of the component in a Natural application is in the following way. A full working example is the program UTIL04 in the library SYSEXPG.

```
define data
local
1 #enum handle of object
1 #files handle of object
1 #file handle of object
end-define
*
* First create an instance of the class SoftwareAG.NaturalX.Enumerator.
create object #enum of 'SoftwareAG.NaturalX.Enumerator.4'
if #enum eq null-handle
  escape routine
end-if
*
* Have a collection of Automation objects
* in the variable #files.
* Code omitted.
* ...
*
* Attach the collection to the enumerator.
send 'Attach' to #enum with #files (ad=o)
*
* Now iterate across the collection.
send 'Next' to #enum return #file
repeat while #file ne null-handle
* Process the item.
* Code omitted.
* ...
* Get the next item.
  send 'Next' to #enum return #file
end-repeat
*
end
```

Interface

The interface of this component, its methods and their usage are described in detail in a separate document.

The component provides the following interface:

- `Interface IEnumerator`

142 Interface IEnumerator

- Purpose 984
- Methods 984

Purpose

This is the main interface of the component SoftwareAG.NaturalX.Enumerator. It is returned when a new instance of the component is created.

```
define data
local
1 #enum handle of object
end-define
*
* Create an instance of the class SoftwareAG.NaturalX.Enumerator.
create object #enum of 'SoftwareAG.NaturalX.Enumerator.4'
if #enum eq null-handle
  escape routine
end-if
*
end
```

After successful execution of the `CREATE OBJECT` statement, the variable `#enum` contains an interface of the type `IEnumerator`.

Methods

The following methods are available:

- Attach
- Reset
- Next

Attach

Attaches a collection to the enumerator. A previously attached collection is then automatically detached. After having attached a collection, the enumerator can be used to enumerate the items contained in that collection.

Parameters

Name	Natural Type	Variant Type	Remark
Collection	HANDLE OF OBJECT	VT_DISPATCH	By value

Collection
A collection of Automation objects.

Reset

Resets the enumerator to its initial state. A subsequent call to the method `Next` returns the first item in the collection.

Next

Returns the next item in the collection. If there is no next item, `NULL-HANDLE` is returned. To start the enumeration over, the method `Reset` can be called.

Parameters

Name	Natural Type	Variant Type	Remark
Return value	HANDLE OF OBJECT	VT_DISPATCH	

Return value
An interface to the next item in the collection.

143 Für Natural reservierte Schlüsselwörter

- Alphabetische Liste der für Natural reservierten Schlüsselwörter 988
- Prüfung auf für Natural reservierte Schlüsselwörter durchführen 1003

Dieses Kapitel enthält eine Liste aller in der Natural-Programmiersprache reservierten Schlüsselwörter.

Wichtig: Um mögliche Namenskonflikte zu vermeiden, empfiehlt es sich sehr, diese für Natural reservierten Schlüsselwörter nicht als Namen für Variablen zu benutzen.

Alphabetische Liste der für Natural reservierten Schlüsselwörter

Die folgende Liste ist eine Übersicht der für Natural reservierten Schlüsselwörter und dient nur der allgemeinen Information. Benutzen Sie in Zweifelsfällen die **Schlüsselwort-Prüffunktion** des Compilers.

[**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**]

- A -

ABS
ABSOLUTE
ACCEPT
ACTION
ACTIVATION
AD
ADD
AFTER
AL
ALARM
ALL
ALPHA
ALPHABETICALLY
AND
ANY
APPL
APPLICATION
ARRAY
AS
ASC
ASCENDING
ASSIGN
ASSIGNING
ASYNC
AT
ATN
ATT

ATTRIBUTES
AUTH
AUTHORIZATION
AUTO
AVER
AVG

- B -

BACKOUT
BACKWARD
BASE
BEFORE
BETWEEN
BLOCK
BOT
BOTTOM
BREAK
BROWSE
BUT
BX
BY

- C -

CABINET
CALL
CALLDBPROC
CALLING
CALLNAT
CAP
CAPTIONED
CASE
CC
CD
CDID
CF
CHAR
CHARLENGTH
CHARPOSITION
CHILD
CIPH
CIPHER
CLASS
CLOSE
CLR

COALESCE
CODEPAGE
COMMAND
COMMIT
COMPOSE
COMPRESS
COMPUTE
CONCAT
CONDITION
CONST
CONSTANT
CONTEXT
CONTROL
CONVERSATION
COPIES
COPY
COS
COUNT
COUPLED
CS
CURRENT
CURSOR
CV

- D -

DATA
DATAAREA
DATE
DAY
DAYS
DC
DECIDE
DECIMAL
DEFINE
DEFINITION
DELETE
DELIMITED
DELIMITER
DELIMITERS
DESC
DESCENDING
DIALOG
DIALOG-ID
DIGITS

DIRECTION
DISABLED
DISP
DISPLAY
DISTINCT
DIVIDE
DL
DLOGOFF
DLOGON
DNATIVE
DNRET
DO
DOCUMENT
DOEND
DOWNLOAD
DU
DY
DYNAMIC

- E -

EDITED
EJ
EJECT
ELSE
EM
ENCODED
END
END-ALL
END-BEFORE
END-BREAK
END-BROWSE
END-CLASS
END-DECIDE
END-DEFINE
END-ENDDATA
END-ENDFILE
END-ENDPAGE
END-ERROR
END-FILE
END-FIND
END-FOR
END-FUNCTION
END-HISTOGRAM
ENDHOC

END-IF
END-INTERFACE
END-LOOP
END-METHOD
END-NOREC
END-PARAMETERS
END-PARSE
END-PROCESS
END-PROPERTY
END-PROTOTYPE
END-READ
END-REPEAT
END-RESULT
END-SELECT
END-SORT
END-START
END-SUBROUTINE
END-TOPPAGE
END-WORK
ENDING
ENTER
ENTIRE
ENTR
EQ
EQUAL
ERASE
ERROR
ERRORS
ES
ESCAPE
EVEN
EVENT
EVERY
EXAMINE
EXCEPT
EXISTS
EXIT
EXP
EXPAND
EXPORT
EXTERNAL
EXTRACTING

- F -

FALSE
FC
FETCH
FIELD
FIELDS
FILE
FILL
FILLER
FINAL
FIND
FIRST
FL
FLOAT
FOR
FORM
FORMAT
FORMATTED
FORMATTING
FORMS
FORWARD
FOUND
FRAC
FRAMED
FROM
FS
FULL
FUNCTION
FUNCTIONS

- G -

GC
GE
GEN
GENERATED
GET
GFID
GIVE
GIVING
GLOBAL
GLOBALS
GREATER
GT
GUI

- H -

HANDLE
HAVING
HC
HD
HE
HEADER
HEX
HISTOGRAM
HOLD
HORIZ
HORIZONTALLY
HOUR
HOURS
HW

- I -

IA
IC
ID
IDENTICAL
IF
IGNORE
IM
IMMEDIATE
IMPORT
IN
INC
INCCONT
INCDIC
INCDIR
INCLUDE
INCLUDED
INCLUDING
INCMAC
INDEPENDENT
INDEX
INDEXED
INDICATOR
INIT
INITIAL
INNER
INPUT
INSENSITIVE

INSERT
INT
INTEGER
INTERCEPTED
INTERFACE
INTERFACE4
INTERMEDIATE
INTERSECT
INTO
INVERTED
INVESTIGATE
IP
IS
ISN

- J -

JOIN
JUST
JUSTIFIED

- K -

KD
KEEP
KEY
KEYS

- L -

LANGUAGE
LAST
LC
LE
LEAVE
LEAVING
LEFT
LENGTH
LESS
LEVEL
LIB
LIBPW
LIBRARY
LIBRARY-PASSWORD
LIKE
LIMIT

LINDICATOR
LINES
LISTED
LOCAL
LOCKS
LOG
LOG-LS
LOG-PS
LOGICAL
LOOP
LOWER
LS
LT

- M -

MACROAREA
MAP
MARK
MASK
MAX
MC
MCG
MESSAGES
METHOD
MGID
MICROSECOND
MIN
MINUTE
MODAL
MODIFIED
MODULE
MONTH
MORE
MOVE
MOVING
MP
MS
MT
MULTI-FETCH
MULTIPLY

- N -

NAME
NAMED

NAMESPACE
NATIVE
NAVER
NC
NCOUNT
NE
NEWPAGE
NL
NMIN
NO
NODE
NOHDR
NONE
NORMALIZE
NORMALIZED
NOT
NOTIT
NOTITLE
NULL
NULL-HANDLE
NUMBER
NUMERIC

- O -

OBJECT
OBTAIN
OCCURRENCES
OF
OFF
OFFSET
OLD
ON
ONCE
ONLY
OPEN
OPTIMIZE
OPTIONAL
OPTIONS
OR
ORDER
OUTER
OUTPUT

- P -

PACKAGESET
PAGE
PARAMETER
PARAMETERS
PARENT
PARSE
PASS
PASSW
PASSWORD
PATH
PATTERN
PA1
PA2
PA3
PC
PD
PEN
PERFORM
PF*n* (*n* = 1 to 9)
PF*nn*(*nn* = 10 to 99)
PGDN
PGUP
PGM
PHYSICAL
PM
POLICY
POS
POSITION
PREFIX
PRINT
PRINTER
PROCESS
PROCESSING
PROFILE
PROGRAM
PROPERTY
PROTOTYPE
PRTY
PS
PT
PW

- Q -

QUARTER
QUERYNO

- R -

RD
READ
READONLY
REC
RECORD
RECORDS
RECURSIVELY
REDEFINE
REDUCE
REFERENCED
REFERENCING
REINPUT
REJECT
REL
RELATION
RELATIONSHIP
RELEASE
REMAINDER
REPEAT
REPLACE
REPORT
REPORTER
REPOSITION
REQUEST
REQUIRED
RESET
RESETTING
RESIZE
RESPONSE
RESTORE
RESULT
RET
RETAIN
RETAINED
RETRY
RETURN
RETURNS
REVERSED
RG

RIGHT
ROLLBACK
ROUNDED
ROUTINE
ROW
ROWS
RR
RS
RULEVAR
RUN

- S -

SA
SAME
SCAN
SCREEN
SCROLL
SECOND
SELECT
SELECTION
SEND
SENSITIVE
SEPARATE
SEQUENCE
SERVER
SET
SETS
SETTIME
SF
SG
SGN
SHORT
SHOW
SIN
SINGLE
SIZE
SKIP
SL
SM
SOME
SORT
SORTED
SORTKEY
SOUND

SPACE
SPECIFIED
SQL
SQLID
SQRT
STACK
START
STARTING
STATEMENT
STATIC
STATUS
STEP
STOP
STORE
SUBPROGRAM
SUBPROGRAMS
SUBROUTINE
SUBSTR
SUBSTRING
SUBTRACT
SUM
SUPPRESS
SUPPRESSED
SUSPEND
SYMBOL
SYNC
SYSTEM

- T -

TAN
TC
TERMINATE
TEXT
TEXTAREA
TEXTVARIABLE
THAN
THEM
THEN
THRU
TIME
TIMESTAMP
TIMEZONE
TITLE
TO

TOP
TOTAL
TP
TR
TRAILER
TRANSACTION
TRANSFER
TRANSLATE
TREQ
TRUE
TS
TYPE
TYPES

- U -

UC
UNDERLINED
UNION
UNIQUE
UNKNOWN
UNTIL
UPDATE
UPLOAD
UPPER
UR
USED
USER
USING

- V -

VAL
VALUE
VALUES
VARGRAPHIC
VARIABLE
VARIABLES
VERT
VERTICALLY
VIA
VIEW

- W -

WH

WHEN
WHERE
WHILE
WINDOW
WITH
WORK
WRITE
WITH_CTE

- X -

XML

- Y -

YEAR

- Z -

ZD
ZP

Prüfung auf für Natural reservierte Schlüsselwörter durchführen

Es gibt eine Untermenge von Natural-Schlüsselwörtern, die zweideutig wären, wenn sie als Namen für Variablen benutzt würden. Dies sind insbesondere Schlüsselwörter, die Natural-Statements (`ADD`, `FIND` usw.) oder System-Funktionen (`ABS`, `SUM` usw.) identifizieren. Wenn Sie ein solches Schlüsselwort als Namen einer Variable benutzen, können Sie diese Variable nicht im Zusammenhang mit optionalen Operanden (mit `CALLNAT`, `WRITE` usw.) benutzen.

Beispiel:

```
DEFINE DATA LOCAL
1 ADD (A10)
END-DEFINE
CALLNAT 'MYSUB' ADD 4     /* ADD is regarded as ADD statement
END
```

Um Variablennamen in einem Programmierobjekt auf solche für Natural reservierten Schlüsselwörter zu überprüfen, können Sie den Natural-Profilparameter `KCHECK` oder die `KCHECK`-Option des Systemkommandos `COMPOPT` benutzen.

Die folgende Tabelle enthält eine Liste der für Natural reservierten Schlüsselwörter, die von `KC` oder `KCHECK` überprüft werden.

A - D	E - F	G - P	R - S	T - W
A-AVER	EJECT	GET	READ	TAN
ABS	ELSE	HISTOGRAM	REDEFINE	TERMINATE
ACCEPT	END	IF	REDUCE	TOP
ADD	END-ALL	IGNORE	REINPUT	TOTAL
ALL	END-BEFORE	IMPORT	REJECT	TRANSFER
A-MAX	END-BREAK	INCCONT	RELEASE	TRUE
A-MIN	END-BROWSE	INCDIC	REPEAT	UNTIL
A-NAVER	END-DECIDE	INCDIR	REQUEST	UPDATE
A-NCOUNT	END-ENDDATA	INCLUDE	RESET	UPLOAD
A-NMIN	END-DECIDE	INCMAC	RESIZE	VAL
ANY	END-ENDDATA	INPUT	RESTORE	VALUE
ASSIGN	END-ENDFILE	INSERT	RET	VALUES
A-SUM	END-ENDPAGE	INT	RETRY	WASTE
AT	END-ERROR	INVESTIGATE	RETURN	WHEN
ATN	END-FILE	LIMIT	ROLLBACK	WHILE
AVER	END-FIND	LOG	RULEVAR	WITH_CTE
BACKOUT	END-FOR	LOOP	RUN	WRITE
BEFORE	END-HISTOGRAM	MAP	SELECT	
BREAK	ENDHOC	MAX	SEND	
BROWSE	END-IF	MIN	SEPARATE	
CALL	END-LOOP	MOVE	SET	
CALLDBPROC	END-NOREC	MULTIPLY	SETTIME	
CALLNAT	END-PARSE	NAVER	SGN	
CLOSE	END-PROCESS	NCOUNT	SHOW	
COMMIT	END-READ	NEWPAGE	SIN	
COMPOSE	END-REPEAT	NMIN	SKIP	
COMPRESS	END-RESULT	NONE	SORT	
COMPUTE	END-SELECT	NULL-HANDLE	SORTKEY	
COPY	END-SORT	OBTAIN	SQRT	
COS	END-START	OLD	STACK	
COUNT	END-SUBROUTINE	ON	START	
CREATE	END-TOPPAGE	OPEN	STOP	
DECIDE	END-WORK	OPTIONS	STORE	
DEFINE	ENTIRE	PARSE	SUBSTR	
DELETE	ESCAPE	PASSW	SUBSTRING	
DISPLAY	EXAMINE	PERFORM	SUBTRACT	
DIVIDE	EXP	POS	SUM	
DLOGOFF	EXPAND	PRINT	SUSPEND	
DLOGON	EXPORT	PROCESS		
DNATIVE	FALSE			
DO	FETCH			
DOEND	FIND			
DOWNLOAD	FOR			
	FORMAT			

A - D	E - F	G - P	R - S	T - W
	FRAC			

Standardmäßig wird keine Schlüsselwort-Prüfung durchgeführt.

144 Referenzierte Beispielprogramme

- READ-Statement 1008
- FIND-Statement 1009
- Geschachtelte READ- und FIND-Statements 1013
- ACCEPT- und REJECT-Statements 1015
- AT START OF DATA- und AT END OF DATA-Statements 1017
- DISPLAY- und WRITE-Statements 1020
- DISPLAY-Statement 1024
- Spaltenüberschriften 1025
- Feldausgabe-relevante Parameter 1027
- Editiermasken 1033
- DISPLAY VERT mit WRITE-Statement 1036
- AT BREAK-Statement 1037
- Statements COMPUTE, MOVE und COMPRESS 1038
- Systemvariablen 1041
- Systemfunktionen 1044

Dieses Kapitel enthält einige zusätzliche Beispielprogramme, auf die im *Leitfaden zur Programmierung* verwiesen wird.

Diese Beispielprogramme befinden sich ebenso wie die übrigen im *Leitfaden zur Programmierung* enthaltenen Beispiele in der Library SYSEXPG.

READ-Statement

Auf das folgende Beispiel wird im Abschnitt *Datenbankzugriffe* verwiesen.

READX03 - READ-Statement (mit LOGICAL-Klausel)

```
** Example 'READX03': READ (with LOGICAL clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 JOB-TITLE
END-DEFINE
*
LIMIT 8
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID
  DISPLAY NOTITLE  *ISN      NAME
                  'PERS-NO'  PERSONNEL-ID
                  'POSITION' JOB-TITLE
END-READ
END
```

Ausgabe des Programms READX03:

```
   ISN              NAME         PERS-NO          POSITION
----------- -------------------- -------- -------------------------

        204 SCHINDLER            11100102 PROGRAMMIERER
        205 SCHIRM               11100105 SYSTEMPROGRAMMIERER
        206 SCHMITT              11100106 OPERATOR
        207 SCHMIDT              11100107 SEKRETAERIN
        208 SCHNEIDER            11100108 SACHBEARBEITER
        209 SCHNEIDER            11100109 SEKRETAERIN
        210 BUNGERT              11100110 SYSTEMPROGRAMMIERER
        211 THIELE               11100111 SEKRETAERIN
```

FIND-Statement

Auf die folgenden Beispiele wird im Abschnitt *Datenbankzugriffe* verwiesen.

FINDX07 - FIND-Statement (mit mehreren Klauseln)

```
** Example 'FINDX07': FIND  (with several clauses)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
FIND EMPLOY-VIEW WITH  PHONETIC-NAME = 'JONES' OR = 'BECKR'
                 AND   CITY          = 'BOSTON' THRU 'NEW YORK'
                 BUT NOT               'CHAPEL HILL'
                 SORTED BY NAME
                 WHERE SALARY (1)    < 28000
  DISPLAY NOTITLE NAME FIRST-NAME CITY SALARY (1)
END-FIND
END
```

Ausgabe des Programms `FINDX07`:

```
        NAME                 FIRST-NAME                CITY            ANNUAL
                                                                       SALARY
-------------------- -------------------- -------------------- ----------

BAKER                PAULINE              DERBY                      4450
JONES                MARTHA               KALAMAZOO                 21000
JONES                KEVIN                DERBY                      7000
```

FINDX08 - FIND-Statement (mit LIMIT)

```
** Example 'FINDX08': FIND (with LIMIT)
**                    Demonstrates FIND statement with LIMIT option to
**                    terminate program with an error message if the
**                    number of records selected exceeds a specified
**                    limit (no output).
************************************************************************
DEFINE DATA LOCAL
```

```
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FIND EMPLOY-VIEW WITH LIMIT (5) JOB-TITLE = 'SALES PERSON'
  DISPLAY NAME JOB-TITLE
END-FIND
END
```

Von Programm `FINDX08` verursachter Laufzeitfehler:

```
NAT1005 More records found than specified in search limit.
```

FINDX09 - FIND-Statement (unter Verwendung von *NUMBER, *COUNTER, *ISN)

```
** Example 'FINDX09': FIND (using *NUMBER, *COUNTER, *ISN)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 DEPT
  2 NAME
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY  = 'BOSTON'
                 WHERE DEPT = 'TECH00' THRU 'TECH10'
  DISPLAY NOTITLE
          'COUNTER' *COUNTER NAME DEPT 'ISN' *ISN
  AT START OF DATA
    WRITE '(TOTAL NUMBER IN BOSTON:' *NUMBER ')' /
  END-START
END-FIND
END
```

Ausgabe des Programms `FINDX09`:

```
  COUNTER           NAME         DEPARTMENT     ISN
                                    CODE
----------- -------------------- ---------- -----------

(TOTAL NUMBER IN BOSTON:           7 )

          1 STANWOOD             TECH10             782
          2 PERREAULT            TECH10             842
```

FINDX10 – FIND-Statement (in Kombination mit READ)

```
** Example 'FINDX10': FIND (combined with READ)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
*
EMP. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  VEH. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    IF NO RECORDS FOUND
      MOVE '*** NO CAR ***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            NAME (EMP.) (IS=ON)
            FIRST-NAME (EMP.) (IS=ON)
            MAKE (VEH.)
  END-FIND
END-READ
END
```

Ausgabe des Programms `FINDX10`:

```
        NAME              FIRST-NAME              MAKE
-------------------- -------------------- --------------------

JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
                                          MG
                     EDWARD               GENERAL MOTORS
                     MARTHA               GENERAL MOTORS
                     LAUREL               GENERAL MOTORS
                     KEVIN                DATSUN
                     GREGORY              FORD
JOPER                MANFRED              *** NO CAR ***
JOUSSELIN            DANIEL               RENAULT
JUBE                 GABRIEL              *** NO CAR ***
JUNG                 ERNST                *** NO CAR ***
```

```
JUNKIN               JEREMY               *** NO CAR ***
KAISER               REINER               *** NO CAR ***
```

FINDX11 - FIND NUMBER-Statement (mit *NUMBER)

```
** Example 'FINDX11': FIND NUMBER (with *NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY        (1)
*
1 #PERCENT        (N.2)
1 REDEFINE #PERCENT
  2 #WHOLE-NBR    (N2)
1 #ALL-BOST       (N3.2)
1 #SECR-ONLY      (N3.2)
1 #BOST-NBR       (N3)
1 #SECR-NBR       (N3)
END-DEFINE
*
F1. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
F2. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
                            AND JOB-TITLE = 'SECRETARY'
*
MOVE *NUMBER(F1.) TO #ALL-BOST  #BOST-NBR
MOVE *NUMBER(F2.) TO #SECR-ONLY #SECR-NBR
DIVIDE #ALL-BOST INTO #SECR-ONLY GIVING #PERCENT
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
  'There are' #BOST-NBR 'employees in the Boston offices.' /
  #SECR-NBR '(=' #WHOLE-NBR (EM=99%')') 'are secretaries.'
*
SKIP 1
FIND EMPLOY-VIEW WITH CITY      = 'BOSTON'
                 AND  JOB-TITLE = 'SECRETARY'
  DISPLAY NAME FIRST-NAME JOB-TITLE SALARY (1)
END-FIND
END
```

Ausgabe des Programms `FINDX11`:

```
There are    7 employees in the Boston offices.
   3 (= 42%) are secretaries.
-------------------------------------------------------------------------------

        NAME              FIRST-NAME              CURRENT            ANNUAL
                                                  POSITION           SALARY
-------------------- -------------------- ------------------------- ----------

SHAW                 LESLIE               SECRETARY                      18000
CREMER               WALT                 SECRETARY                      20000
COHEN                JOHN                 SECRETARY                      16000
```

Geschachtelte READ- und FIND-Statements

Auf die folgenden Beispiele wird im Abschnitt *Datenbank-Verarbeitungsschleifen* verwiesen.

READX04 – READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)

```
** Example 'READX04': READ (in combination with FIND and the system
**                    variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

Ausgabe des Programms READX04:

```
  CNT          NAME       FIRST-NAME   NMBR      CNT            MAKE
--------- -------------- ---------- --------- --------- --------------------

        1 JONES          VIRGINIA           1         1 CHRYSLER
        2 JONES          MARSHA             2         1 CHRYSLER
        2 JONES          MARSHA             2         2 CHRYSLER
        3 JONES          ROBERT             1         1 GENERAL MOTORS
        4 JONES          LILLY              2         1 FORD
        4 JONES          LILLY              2         2 MG
        5 JONES          EDWARD             1         1 GENERAL MOTORS
        6 JONES          MARTHA             1         1 GENERAL MOTORS
        7 JONES          LAUREL             1         1 GENERAL MOTORS
        8 JONES          KEVIN              1         1 DATSUN
        9 JONES          GREGORY            1         1 FORD
       10 JOPER          MANFRED            0         0
```

LIMITX01 - LIMIT-Statement (für READ- und FIND-Schleifenverarbeitung)

```
** Example 'LIMITX01': LIMIT  (for READ, FIND loop processing)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 4
*
READ EMPLOY-VIEW BY NAME STARTING FROM 'A'
  FIND VEH-VIEW WITH PERSONNEL-ID = EMPLOY-VIEW.PERSONNEL-ID
    IF NO RECORDS FOUND
      MOVE 'NO CAR' TO MAKE
    END-NOREC
    DISPLAY PERSONNEL-ID NAME FIRST-NAME MAKE
  END-FIND
END-READ
END
```

Ausgabe des Programms `LIMITX01`:

```
Page      1                                                04-12-13  14:01:57

PERSONNEL-ID         NAME                 FIRST-NAME              MAKE
------------ -------------------- -------------------- --------------------

             ABELLAN              KEPA                 NO CAR
30000231     ACHIESON             ROBERT               FORD
             ADAM                 SIMONE               NO CAR
20008800     ADKINSON             JEFF                 GENERAL MOTORS
```

ACCEPT- und REJECT-Statements

Auf die folgenden Beispiele wird im Abschnitt ***Datensätze mit ACCEPT / REJECT auswählen*** verwiesen.

ACCEPX04 - ACCEPT IF ... LESS THAN ...-Statement

```
** Example 'ACCEPX04': ACCEPT IF ... LES THAN ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  ACCEPT IF SALARY (1) LESS THAN 38000
  DISPLAY NOTITLE PERSONNEL-ID NAME JOB-TITLE SALARY (1)
END-READ
END
```

Ausgabe des Programms `ACCEPX04`:

```
PERSONNEL         NAME                   CURRENT            ANNUAL
   ID                                   POSITION            SALARY
--------- -------------------- ------------------------- ----------

20017000  CREMER               ANALYST                        34000
20017100  MARKUSH              TRAINEE                        22000
20017400  NEEDHAM              PROGRAMMER                     32500
20017500  JACKSON              PROGRAMMER                     33000
20017600  PIETSCH              SECRETARY                      22000
```

```
20017700  PAUL                  SECRETARY                      23000
20018000  FARRIS                PROGRAMMER                     30500
20018100  EVANS                 PROGRAMMER                     31000
20018200  HERZOG                PROGRAMMER                     31500
20018300  LORIE                 SALES PERSON                   28000
20018400  HALL                  SALES PERSON                   30000
20018500  JACKSON               MANAGER                        36000
20018800  SMITH                 SECRETARY                      24000
20018900  LOWRY                 SECRETARY                      25000
```

ACCEPX05 – ACCEPT IF ... AND ...-Statement

```
** Example 'ACCEPX05': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:2)
END-DEFINE
*
LIMIT 6
READ EMPLOY-VIEW PHYSICAL WHERE SALARY(2) > 0
  ACCEPT IF  SALARY(1) > 10000
         AND SALARY(1) < 50000
  DISPLAY (AL=15) 'SALARY I' SALARY (1) 'SALARY II'  SALARY (2)
                   NAME JOB-TITLE  CITY
END-READ
END
```

Ausgabe des Programms ACCEPX05:

```
Page      1                                                04-12-13  14:05:28

 SALARY I  SALARY II       NAME           CURRENT          CITY
                                          POSITION
---------- ---------- --------------- --------------- ---------------

    48000      46000 SPENGLER        SACHBEARBEITER  DARMSTADT
    45000      40000 SPECK           SACHBEARBEITER  DARMSTADT
    48000      46000 SCHINDLER       PROGRAMMIERER   HEPPENHEIM
    36000      32000 SCHMIDT         SEKRETAERIN     HEPPENHEIM
```

ACCEPX06 – REJECT IF ... OR ...-Statement

```
** Example 'ACCEPX06': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY (1)
  2 JOB-TITLE
  2 CITY
  2 NAME
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '20017000'
  REJECT IF SALARY (1) < 20000
         OR SALARY (1) > 26000
  DISPLAY NOTITLE SALARY (1) NAME JOB-TITLE CITY
END-READ
END
```

Ausgabe des Programms `ACCEPX06`:

```
  ANNUAL           NAME                  CURRENT                CITY
  SALARY                                POSITION
---------- -------------------- ------------------------- --------------------

     22000 MARKUSH              TRAINEE                   LOS ANGELES
     22000 PIETSCH              SECRETARY                 VISTA
     23000 PAUL                 SECRETARY                 NORFOLK
     24000 SMITH                SECRETARY                 SILVER SPRING
     25000 LOWRY                SECRETARY                 LEXINGTON
```

AT START OF DATA- und AT END OF DATA-Statements

Auf die folgenden Beispiele wird im Abschnitt *AT START/END OF DATA-Statements* verwiesen.

ATENDX01 – AT END OF DATA-Statement

```
** Example 'ATENDX01': AT END OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
READ (6) EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE NAME JOB-TITLE
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
END-READ
END
```

Ausgabe des Programms `ATENDX01`:

```
        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

CREMER               ANALYST
MARKUSH              TRAINEE
GEE                  MANAGER
KUNEY                DBA
NEEDHAM              PROGRAMMER
JACKSON              PROGRAMMER

LAST PERSON SELECTED: JACKSON
```

ATSTAX02 – AT START OF DATA-Statement

```
** Example 'ATSTAX02': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY    (1)
  2 CURR-CODE (1)
  2 BONUS     (1,1)
END-DEFINE
*
```

```
LIMIT 3
FIND EMPLOY-VIEW WITH CITY = 'MADRID'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1) CURR-CODE (1)
  /*
  AT START OF DATA
    WRITE NOTITLE *DAT4E /
  END-START
END-FIND
END
```

Ausgabe des Programms `ATSTAX02`:

```

        NAME                 FIRST-NAME          ANNUAL      BONUS     CURRENCY
                                                 SALARY                  CODE
-------------------- -------------------- ---------- ---------- --------

13/12/2004

DE JUAN              JAVIER                        1988000          0 PTA
DE LA MADRID         ANSELMO                       3120000          0 PTA
PINERO               PAULA                         1756000          0 PTA
```

WRITEX09 – WRITE-Statement (in Kombination mit AT END OF DATA)

```

** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX09`:

```
        NAME             DATE              CURRENT
                          OF              POSITION
                         BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                     DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                     DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                     DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY- und WRITE-Statements

Auf die folgenden Beispiele wird im Abschnitt ***Statements DISPLAY und WRITE*** verwiesen.

DISPLX13 – DISPLAY-Statement (zum Vergleich mit WRITEX08 mit WRITE)

```
** Example 'DISPLX13': DISPLAY (compare with WRITEX08 using WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  DISPLAY 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
          '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX13`:

```
Page      1                                                  04-12-13  14:11:28

  PERS           NAME            ANNUAL       BONUS         CITY
   ID         FIRST-NAME         SALARY
-------- ------------------- ------------ ---------- --------------


20027000 CUMMINGS            **      41000       1500 CHAPEL HILL
         PUALA                       38900

20000200 WOOLSEY             **      26000       3000 CHAPEL HILL
         LOUISE                      24700
```

WRITEX08 – WRITE-Statement (zum Vergleich mit DISPLX13 mit DISPLAY)

```
** Example 'WRITEX08': WRITE  (compare with DISPLX13 using DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  WRITE 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
        '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX08`:

```
Page      1                                                  04-12-13  14:12:43

PERS/ID 20027000 CUMMINGS
PUALA               ** ANNUAL SALARY:      41000      38900 BONUS       1500
CHAPEL HILL

PERS/ID 20000200 WOOLSEY
LOUISE              ** ANNUAL SALARY:      26000      24700 BONUS       3000
CHAPEL HILL
```

DISPLX14 – DISPLAY-Statement (mit AL, SF und *n*X)

```
** Example 'DISPLX14': DISPLAY  (with AL, SF and nX)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
  2 CITY
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'W'
  DISPLAY (AL=15 SF=5) NAME CITY / ADDRESS-LINE(1) 2X TELEPHONE
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX14`:

```
Page      1                                                    04-12-13  14:14:00

     NAME                CITY                         TELEPHONE
                        ADDRESS
                                         AREA                TELEPHONE
                                         CODE
---------------     ---------------  ---------------     ---------------

WABER               HEIDELBERG       06221               456452
                    ERBACHERSTR. 78

WADSWORTH           DERBY            0332                515365
                    56 PINECROFT CO

WAGENBACH           FRANKFURT        069                 983218
                    BECKERSTR. 4
```

WRITEX09 – WRITE-Statement (in Kombination mit AT END OF DATA)

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX09`:

```
        NAME             DATE              CURRENT
                          OF              POSITION
                         BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                     DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                     DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                     DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY-Statement

Auf folgendes Beispiel wird im Abschnitt *Seitenüberschriften, Seitenvorschübe und Leerzeilen* verwiesen.

DISPLX21 – DISPLAY-Statement (mit Schrägstrich '/' und zum Vergleich mit WRITE)

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX21`:

```
14:15:50.1     PEOPLE LIVING IN SALT LAKE CITY                     PAGE:      1
               AS OF 13/12/2004


-------------------------------------------------------------------------------
        NAME                 HOME                STREET
     FIRST-NAME              CITY              OR BOX NO.
-------------------- -------------------- --------------------


ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY
```

```
SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                  REGISTER OF
                                SALT LAKE CITY
-------------------------------------------------------------------------------
```

Spaltenüberschriften

Auf das folgende Beispiel wird im Abschnitt *Spaltenüberschriften* verwiesen.

DISPLX15 – DISPLAY-Statement (mit FC, UC)

```
** Example 'DISPLX15': DISPLAY  (with FC, UC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
FORMAT AL=12 GC== UC=%
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'R'
  DISPLAY NOTITLE (FC=*)
          NAME FIRST-NAME CITY  (FC=- UC=-) /
          ADDRESS-LINE(1) TELEPHONE
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX15`:

```
****NAME**** *FIRST-NAME* ----CITY---- ========TELEPHONE========
                          **ADDRESS***
                                       ****AREA**** *TELEPHONE**
                                       ****CODE****
%%%%%%%%%%%% %%%%%%%%%%%% ------------ %%%%%%%%%%%% %%%%%%%%%%%%

RACKMANN     MARIAN       FRANKFURT    069          375849
```

```
                          FINKENSTR. 1

RAMAMOORTHY  TY           SEPULVEDA    209          175-1885
                          12018 BROOKS

RAMAMOORTHY  TIMMIE       SEATTLE      206          151-4673
                          921-178TH PL
```

DISPLX16 – DISPLAY-Statement (mit '/', '`text`', '`text/text`')

```
** Example 'DISPLX16': DISPLAY  (with '/', 'text', 'text/text')
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
READ (5) EMPLOY-VIEW BY NAME STARTING FROM 'E'
  DISPLAY NOTITLE
    '/'          NAME       (AL=12) /* suppressed header
    'FIRST/NAME' FIRST-NAME (AL=10) /* two-line user-defined header
    'ADDRESS'    CITY /             /* user-defined header
    ' '          ADDRESS-LINE(1)    /* 'blank' header
                 TELEPHONE (HC=L)   /* default header
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX16`:

```
               FIRST          ADDRESS        TELEPHONE
                NAME
                                             AREA   TELEPHONE
                                             CODE
             ---------- -------------------- ------ ---------------

EAVES        TREVOR     DERBY                0332   657623
                        17 HARTON ROAD

ECKERT       KARL       OBERRAMSTADT         06154  99722
                        FORSTWEG 22

ECKHARDT     RICHARD    DARMSTADT
```

```
                        BRESLAUERPL. 4

EDMUNDSON    LES        TULSA                918    945-4916
                        2415 ALSOP CT.

EGGERT       HERMANN    STUTTGART            0711   981237
                        RABENGASSE 8
```

Feldausgabe-relevante Parameter

Auf die folgenden Beispiele wird im Abschnitt ***Parameter zur Beeinflussung der Ausgabe von Feldern*** verwiesen.

Sie stehen zur Verfügung, um die Benutzung der Parameter LC, IC, TC, AL, NL, IS, ZP und ES und des Statements SUSPEND IDENTICAL SUPPRESS zu demonstrieren:

DISPLX17 - DISPLAY-Statement (mit NL, AL, IC, LC, TC)

```
** Example 'DISPLX17': DISPLAY (with NL, AL, IC, LC, TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  DISPLAY NOTITLE (IS=ON NL=15)
                    NAME
    '-' '='         FIRST-NAME (AL=12)
    'ANNUAL SALARY' SALARY(1)  (LC=USD   TC=.00)   /
    '+ BONUSES'     BONUS(1,1) (IC='+  ' TC=.00)
  SKIP 1
END-READ
END
```

Ausgabe des Programms DISPLX17:

```
       NAME           FIRST-NAME       ANNUAL SALARY
                                         + BONUSES
-------------------- ------------- ---------------------

JONES               - VIRGINIA     USD           46000.00
                                               +  9000.00

                    - MARSHA       USD           50000.00
                                                  +  0.00

                    - ROBERT       USD           31000.00
                                                  +  0.00
```

DISPLX18 – DISPLAY-Statement (Benutzung von Voreinstellungen für SF, AL, UC, LC, IC, TC und zum Vergleich mit DISPLX19)

```
** Example 'DISPLX18': DISPLAY  (using default settings for SF, AL, UC,
**                       LC, IC, TC and compare with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 BONUS     (1,1)
END-DEFINE
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1)
END-FIND
END
```

Ausgabe des Programms DISPLX18:

```
Page      1                                                04-12-13  14:20:48

       NAME              FIRST-NAME        ANNUAL     BONUS
                                           SALARY
-------------------- -------------------- ---------- ----------

KESSLER             CLARE                     41000          0
ADKINSON            DAVID                     24000          0
GEE                 TOMMIE                    39500          0
HERZOG              JOHN                      31500          0
QUILLION            TIMOTHY                   30500          0
CUMMINGS            PUALA                     41000       1500
```

DISPLX19 - DISPLAY-Statement (mit SF, AL, LC, IC, TC und zum Vergleich mit DISPLX18)

```
** Example 'DISPLX19': DISPLAY  (with SF, AL, LC, IC, TC and compare
**                      with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
FORMAT SF=3 AL=15 UC==
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY (NL=10)
    NAME
    FIRST-NAME  (LC='- ' UC=-)
    SALARY (1)  (LC=USD)
    BONUS (1,1) (IC='*** ' TC=' ***')
END-FIND
END
```

Ausgabe des Programms `DISPLX19`:

```
Page      1                                                04-12-13  14:21:57

     NAME             FIRST-NAME           ANNUAL              BONUS
                                           SALARY
===============   -----------------   ==============   ===================

KESSLER           - CLARE             USD      41000            *** 0 ***
ADKINSON          - DAVID             USD      24000            *** 0 ***
GEE               - TOMMIE            USD      39500            *** 0 ***
HERZOG            - JOHN              USD      31500            *** 0 ***
QUILLION          - TIMOTHY           USD      30500            *** 0 ***
CUMMINGS          - PUALA             USD      41000         *** 1500 ***
```

SUSPEX01 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP bei DISPLAY)

```
** Example 'SUSPEX01': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                     parameters IS, ES, ZP in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

Ausgabe des Programms SUSPEX01:

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
                                CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
                                MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
```

```
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

SUSPEX02 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP in DISPLAY). Identisch mit SUSPEX01, aber mit IS=OFF.

```
** Example 'SUSPEX02': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                     parameters IS, ES, ZP in DISPLAY)
**                     Identical to SUSPEX01, but with IS=OFF.
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=OFF ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

Ausgabe des Programms `SUSPEX02`:

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
```

```
JONES           LILLY           MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

COMPRX03 - COMPRESS-Statement (in Verbindung mit LC und TC)

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
          'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
          '/'               #FULL-SALARY                  /
          '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

Ausgabe des Programms `COMPRX03`:

```
       NAME              FIRST-NAME               JOB DESCRIPTION
-------------------- -------------------- -----------------------------------

SHAW                 LESLIE               JOB     : SECRETARY
                                          SALARY  : USD 18000
                                          VACATION: 2DAYS

STANWOOD             VERNON               JOB     : PROGRAMMER
                                          SALARY  : USD 31000
                                          VACATION: 1DAYS

CREMER               WALT                 JOB     : SECRETARY
                                          SALARY  : USD 20000
                                          VACATION: 3DAYS
```

Editiermasken

Auf folgende Beispiele wird im Abschnitt *Editiermasken – der EM-Parameter* verwiesen.

EDITMX03 - Editiermaske (unterschiedliche EM-Angabe bei alphanumerischen Feldern)

```
** Example 'EDITMX03': Edit mask (different EM for alpha-numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 SALARY(1)
END-DEFINE
*
LIMIT 3
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY 'N A M E'  NAME       (EM=X^^X^^X^^X^^X^^X^^X^^X^^X^^X) /
          'NAME HEX' NAME       (EM=H^H^H^H^H^H^H^H^H^H)
                     FIRST-NAME (EM=' - 'X(15)*)
                     CITY       (EM=X..X(10))
  SKIP 1
END-READ
END
```

Ausgabe des Programms `EDITMX03`:

```
Page      1                                             04-12-13  14:26:57

            N A M E                  FIRST-NAME         CITY
            NAME HEX
-------------------------------- ----------------- ------------

L  O  R  I  E                     - JEAN-PAUL      * C..LEVELAND
D3 D6 D9 C9 C5 40 40 40 40 40 40

H  A  L  L                        - ARTHUR         * A..NN ARBER
C8 C1 D3 D3 40 40 40 40 40 40 40

V  A  S  W  A  N  I               - TOMMIE         * M..ONTERREY
E5 C1 E2 E6 C1 D5 C9 40 40 40 40
```

EDITMX04 - Editiermaske (unterschiedliche EM-Angaben bei numerischen Feldern)

```
** Example 'EDITMX04': Edit mask (different EM for numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
  2 LEAVE-DUE
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW BY PERSONNEL-ID = '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY (SF=4)
          'N A M E'    NAME
          'SALARY'     SALARY(1)  (EM=*USD^ZZZ,999)
          'BONUS (ZZ)' BONUS(1,1) (EM=S*ZZZ,999) /
          'BONUS (Z9)' BONUS(1,1) (EM=SZ99,999+) /
          '->' '='     BONUS(1,1) (EM=-999,999)
          'VAC/DUE'    LEAVE-DUE  (EM=+999)
  SKIP 1
END-READ
END
```

Ausgabe des Programms EDITMX04:

```
Page      1                                               04-12-13  14:27:43

      N A M E             SALARY       BONUS (ZZ)     VAC
                                       BONUS (Z9)     DUE
                                          BONUS
--------------------    -----------    -----------    ---

LORIE                   USD *28,000    +**4,000       +13
                                       + 04,000+
                                       ->  004,000

HALL                    USD *30,000    +**5,000       +14
                                       + 05,000+
                                       ->  005,000
```

EDITMX05 - Editiermaske (EM-Angaben für Datums- und Uhrzeit-Systemvariablen)

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

Ausgabe des Programms `EDITMX05`:

```



DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004           OR   12/13/2004
     JULIAN   : 2004348              OR   2004348
     GREGORIAN: 13.December2004      OR   13December 2004


TIME INTERNAL : 14:28:49             OR   14:28:49.1
              : 14.28.49.1
```

```
              : 02.28.49 PM
              : 14
```

DISPLAY VERT mit WRITE-Statement

WRITEX10 - WRITE-Statement (mit *n*T, T**field* und P**field*)

```
** Example 'WRITEX10': WRITE (with nT, T*field and P*field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 JOB-TITLE
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH JOB-TITLE FROM 'SALES PERSON'
  DISPLAY NOTITLE NAME 30T JOB-TITLE
          VERT AS 'SALARY/BONUS' SALARY(1) BONUS(1,1)
  AT BREAK OF JOB-TITLE
    WRITE 20T 'AVERAGE' T*JOB-TITLE OLD(JOB-TITLE) (AL=15)
              '(SAL)'   P*SALARY AVER(SALARY(1))  /
          46T '(BON)'   P*BONUS  AVER(BONUS(1,1)) /
  END-BREAK
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX10`:

```
        NAME                          CURRENT           SALARY
                                     POSITION           BONUS
--------------------         ------------------------- ----------

SAMUELSON                    SALES PERSON                   32000
                                                             6000

PAPAYANOPOULOS               SALES PERSON                   34000
                                                             7000

HELL                         SALES PERSON                   38000
                                                             9000

                   AVERAGE   SALES PERSON    (SAL)          34666
                                             (BON)           7333
```

AT BREAK-Statement

Auf das folgende Beispiel wird im Abschnitt *Gruppenwechsel* verwiesen.

ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                       MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

Ausgabe des Programms `ATBREX06`:

```
                        -- SALARY STATISTICS BY CITY --

        CITY         SALARY (1)               SALARY (2)
-------------------- ----------               ----------

NEW YORK                  17000                    16100
NEW YORK                  38000                    34900

             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000       -   MIN:      16100
            -  AVER:      27500       -  AVER:      25500
                      2 RECORDS                 2 RECORDS
```

```
            -  NMIN:       17000       -  NMIN:       16100
            - NAVER:       27500       - NAVER:       25500
                      2 RECORDS                 2 RECORDS
```

Statements COMPUTE, MOVE und COMPRESS

Auf die folgenden Beispiele wird im Abschnitt *Datenberechnungen* verwiesen.

WRITEX11 – WRITE-Statement (mit *n*X, *n*/*n* und COMPRESS)

```
** Example 'WRITEX11': WRITE (with nX, n/n and COMPRESS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY       (1)
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 ZIP
  2 CURR-CODE    (1)
  2 JOB-TITLE
  2 LEAVE-DUE
  2 ADDRESS-LINE (1)
*
1 #SALARY        (A8)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
1 #FULL-SALARY   (A25)
1 #VACATION      (A16)
END-DEFINE
*
READ (3) EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '2001800'
  MOVE SALARY(1) TO  #SALARY
  COMPRESS FIRST-NAME NAME                    INTO #FULL-NAME
  COMPRESS ZIP        CITY                    INTO #FULL-CITY
  COMPRESS 'SALARY   :'  CURR-CODE(1)  #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:'  LEAVE-DUE    'DAYS'   INTO #VACATION
  /*
  DISPLAY NOTITLE 'NAME AND ADDRESS' NAME
               5X 'PERS-NO.'         PERSONNEL-ID
               3X 'JOB TITLE'        JOB-TITLE (LC='JOB     : ')
  WRITE   1/5 #FULL-NAME        1/37  #FULL-SALARY
          2/5 ADDRESS-LINE(1)   2/37  #VACATION
          3/5 #FULL-CITY
  SKIP 1
```

```
END-READ
END
```

Ausgabe des Programms `WRITEX11`:

```
  NAME AND ADDRESS       PERS-NO.                JOB TITLE
--------------------     --------   -----------------------------------

FARRIS                   20018000   JOB     : PROGRAMMER
    JACKIE FARRIS                   SALARY  : USD 30500
    918 ELM STREET                  VACATION: 10 DAY
    32306 TALLAHASSEE

EVANS                    20018100   JOB     : PROGRAMMER
    JO EVANS                        SALARY  : USD 31000
    1058 REDSTONE LANE              VACATION: 11 DAY
    68508 LINCOLN

HERZOG                   20018200   JOB     : PROGRAMMER
    JOHN HERZOG                     SALARY  : USD 31500
    255 ZANG STREET #253            VACATION: 12 DAY
    27514 CHAPEL HILL
```

IFX03 - IF-Statement

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
  IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
    MOVE 'CATALOG I'         TO #TEXT
  END-IF
  /*
```

```
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `IFX03`:

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                         ----------
                INCOME:       56000   CATALOGS I AND II
                ===================

RICHMOND                       9150
                                  0
                         ----------
                INCOME:        9150   CATALOG I
                ===================

MONKTON                       13500
                                600
                         ----------
                INCOME:       14100   CATALOG I
                ===================
```

COMPRX03 - COMPRESS-Statement (mit Parametern LC and TC)

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
```

```
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
          'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
          '/'               #FULL-SALARY                  /
          '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

Ausgabe des Programms `COMPRX03`:

```

        NAME                 FIRST-NAME                 JOB DESCRIPTION
-------------------- -------------------- -----------------------------------

SHAW                 LESLIE                JOB     : SECRETARY
                                           SALARY  : USD 18000
                                           VACATION: 2DAYS

STANWOOD             VERNON                JOB     : PROGRAMMER
                                           SALARY  : USD 31000
                                           VACATION: 1DAYS

CREMER               WALT                  JOB     : SECRETARY
                                           SALARY  : USD 20000
                                           VACATION: 3DAYS
```

Systemvariablen

Auf die folgenden Beispiele wird im Abschnitt *Systemvariablen und Systemfunktionen* verwiesen.

EDITMX05 – Editiermaske (EM bei Datums- und Uhrzeit-Systemvariablen)

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

Ausgabe des Programms `EDITMX05`:

```



DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004            OR   12/13/2004
     JULIAN   : 2004348               OR   2004348
     GREGORIAN: 13.December2004       OR   13December 2004


TIME INTERNAL : 14:36:58              OR   14:36:58.8
              : 14.36.58.8
              : 02.36.58 PM
              : 14
```

READX04 - READ-Statement (in Verbindung mit FIND und den Systemvariablen *NUMBER und *COUNTER)

```
** Example 'READX04': READ (in combination with FIND and the system
**                      variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
```

```
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

Ausgabe des Programms `READX04`:

```
  CNT          NAME       FIRST-NAME   NMBR      CNT            MAKE
--------- --------------- ---------- --------- --------- --------------------


        1 JONES           VIRGINIA           1         1 CHRYSLER
        2 JONES           MARSHA             2         1 CHRYSLER
        2 JONES           MARSHA             2         2 CHRYSLER
        3 JONES           ROBERT             1         1 GENERAL MOTORS
        4 JONES           LILLY              2         1 FORD
        4 JONES           LILLY              2         2 MG
        5 JONES           EDWARD             1         1 GENERAL MOTORS
        6 JONES           MARTHA             1         1 GENERAL MOTORS
        7 JONES           LAUREL             1         1 GENERAL MOTORS
        8 JONES           KEVIN              1         1 DATSUN
        9 JONES           GREGORY            1         1 FORD
       10 JOPER           MANFRED            0         0
```

WTITLX01 – WRITE TITLE-Statement (mit *PAGE–NUMBER)

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
```

```
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

Ausgabe des Programms `WTITLX01`:

```
YEAR:  1980               PAGE      1
YEAR          MAKE         MAINT-COST
----- -------------------- ----------

 1980 RENAULT                  20000
 1980 RENAULT                  20000
 1980 PEUGEOT                  20000
```

Systemfunktionen

Auf die folgenden Beispiele wird im Abschnitt *Systemvariablen und Systemfunktionen* verwiesen.

ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                     MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
```

```
     14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
     13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
     13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
     16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
     13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
     13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
     16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

Ausgabe des Programms ATBREX06:

```
                        -- SALARY STATISTICS BY CITY --

        CITY         SALARY (1)               SALARY (2)
-------------------- ----------               ----------

NEW YORK                  17000                    16100
NEW YORK                  38000                    34900

             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000      -   MIN:      16100
            -  AVER:      27500      -  AVER:      25500
                      2 RECORDS                2 RECORDS

            -  NMIN:      17000      -  NMIN:      16100
            - NAVER:      27500      - NAVER:      25500
                      2 RECORDS                2 RECORDS
```

ATENPX01 - AT END OF PAGE-Statement (mit der durch GIVE SYSTEM FUNCTIONS in DISPLAY verfügbaren Systemfunktion)

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT END OF PAGE
```

```
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

Ausgabe des Programms ATENPX01:

```
        NAME                  CURRENT           SALARY
                             POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                        34000
MARKUSH              TRAINEE                        22000
GEE                  MANAGER                        39500
KUNEY                DBA                            40200
NEEDHAM              PROGRAMMER                     32500
JACKSON              PROGRAMMER                     33000
PIETSCH              SECRETARY                      22000
PAUL                 SECRETARY                      23000
HERZOG               MANAGER                        48500
DEKKER               DBA                            48000

                       AVERAGE SALARY: ...      34270
```